Data Structures, Minimization and Complexity of Boolean Functions

A Thesis

Submitted to the College of Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in the

Department of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan

Canada

by Yuke Wang

September, 1995

The author claims copyright. Use shall not be made of the material contained herein without proper acknowledgment, as indicated on the following page.

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-23956-X

Canadä

UNIVERSITY OF SASKATCHEWAN
College of Graduate Studies and Research

SUMMARY OF DISSERTATION

Submitted in partial fulfilment
of the requirements for the

DEGREE OF DOCTOR OF PHILOSOPHY

by

Yuke Wang

Department of Computer Science
University of Saskatchewan

September 1995

Examining Committee:

Dr. M. Stoneham	Dean's Designate, Chair College of Graduate Studies and Research
Dr. J. Tremblay	Chair of Advisory Committee, Department of Computer Science
Dr. C. McCrosky	Supervisor, Dept. of Computer Science.
Dr. M. Keil	Department of Computer Science.
Dr. E. Neufeld	Department of Computer Science.
Dr. K. Taylor	Department of Mathematics.

External Examiner:

Dr. M. Aboulhamid,
Departement d'Informatique et de Recherche Operationnelle
Universite de Montreal
Montreal, Quebec, CANADA

Data Structures, Minimization and Complexity of Boolean Functions

Boolean function manipulation is an important component of computer science. This thesis presents results related to Boolean function representation and minimization. The Boolean function minimization problem is re-defined. A new Boolean function classification theory based on permutation and extension is developed. More efficient Boolean function minimization algorithms can be derived based on the classification theory. The thesis also examines the complexity issues and graph structures of OBDDs of some special Boolean functions. Moreover, some new data structures for Boolean functions are reported in this thesis. Some functions are found to have constant complexity in the new data structure while having exponential complexity in existing data structures.

DISTRIBUTION NOTICE

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan,

Canada S7N 0W0

Data Structures, Minimization and Complexity of Boolean Functions

Candidate: Yuke Wang

Supervisor: Carl D. McCrosky

Doctor of Philosophy Thesis

Presented to the College of Graduate Studies

University of Saskatchewan

September 1995

Abstract

Boolean function manipulation is an important component of computer science. This thesis presents results related to Boolean function representation and minimization. The Boolean function minimization problem is re-defined. A new Boolean function *classification theory* based on permutation and extension is developed. More efficient Boolean function *minimization algorithms* can be derived based on the classification theory. The thesis also examines the *complexity issues* and *graph structure* of OBDDs of some special Boolean functions. Moreover, some *new data structures* for Boolean functions are reported in this thesis. Some functions are found to have constant complexity in the new data structure while having exponential complexity in existing data structures.

Acknowledgments

I have gone through a long way, both geographically and mentally, from my countryside home town in China to the time I finish this thesis. Left behind along the way is much happiness and sadness, pain and sweetness. Carrying on with me will be the mission started by the God and vivid appreciation towards numerous people who have helped me during my journey. I will always remember these people:

Professor C. McCrosky and Mrs. J. McCrosky, Prof. M. Keil, Prof. E. Neufeld, Prof. K. Taylor, Prof. M. Aboulmaid, Prof. M. Abd-el-Barr, Prof. D. Eager, Prof. J. Lewis, Prof. G. McCalla, Ms. M. Desjardins, Ms. G. Worker, Mr. D. Lynch, Mr. K. Cliff. Mr. and Mrs. Xia.

In particular, I thank Professor C. McCrosky for many discussions and support through many years. I thank Mr. C. Klein for kindly supporting my work by making accessible the Macintosh computers of TRlabs of Saskatoon. Financial support was provided by a scholarship from the University of Saskatchewan.

This work is dedicated to my mother, who will continue to bless me, as she always did, on earth or in heaven.

TABLE OF CONTENTS

Chapter 1 Introduction 1

§1.1 Motivation and Objectives 1

§1.2 Overview of the Thesis 5

Part I Background 7

Chapter 2 Boolean Functions, Their Representations and Minimization 8

§2.1 Permutation 8

§2.2 Boolean Functions and Their Representations 10

§2.3 Permutations on Boolean Functions 19

Chapter 3 OBDD Review 21

§3.1 OBDDs 21

§3.2 Partial-OBDDs 24

§3.3 Permutation and Minimization of OBDDs 27

Part II Boolean Function Classification, Minimization, and Application 33

Chapter 4 Single-Faced Boolean Function and the Structure of Their OBDDs 34

§4.1 A Boolean Function Classification Theory 34

§4.2 A Refined Classification Theory 37

§4.3 The Structure of Single-faced Functions. 42

§4.4 Minimization of Single-faced Functions 46

Chapter 5 OBDD Structure and Complexity of Symmetry Boolean Functions 52

§5.1 Introduction 52

§5.2 The Structure of Symmetric Functions 53

§5.3 Symmetry Detection Algorithm 59

Part III New Data Structures 61

Chapter 6 Some Hard Examples and More Data Structures 62
§6.1 Hard Examples 62
§6.2 Free BDDs 64
Chapter 7 A Unified Data Structure 67
§7.1 Introduction 68
§7.2 Integer Lists and Their Representations 69
§7.2.1 Integer Lists 69
§7.2.2 The Representation of Single Non-monotonic Lists 71
§7.2.3 The Representation of Multiple Non-monotonic Lists 75
§7.2.4 Representation of Piecewise Monotonic Lists 76
§7.3 Integer List Representation of Boolean Functions 77
§7.3.1 Integer Lists and Boolean Functions 77
§7.3.2 Default Representation of Boolean Functions 79
§7.3.3 Piecewise Monotonic Representation of Boolean Functions 82
§7.4 The Power of Difference Lists 83
§7.5 Operations on D-Lists 89
§7.6 Conclusion 92
Chapter 8 Conclusion 94
References 96

List of Figures

Figure 2.1 Truth table and the corresponding complete binary decision tree 12

Figure 3.1 OBDD of $f_{V_1} = x_1x_2 + x_3x_4 + x_5x_6$ 22

Figure 3.2 The OBDD and partial-OBDD of $x_1x_2 + x_3x_4 + x_5x_6$ 25

Figure 3.3 The OBDD of $x_1x_2 + x_3x_4 + x_5x_6$ under different variable orderings.......... 27

Figure 3.4 Partial-OBDDs of F(3) under different orderings 28

Figure 3.5 OBDDs with identical graphs which are not isomorphic 31

Figure 3.6 Permutation of OBDDs 31

Figure 4.1 The root graph of the p-OBDD of a 0-single-faced function 43

Figure 4.2 Three kinds of paths in the OBDD of a 1-single-faced function 44

Figure 4.3 The root graph of the p-OBDD of a 1-single-faced function 45

Figure 4.4 An example OBDD of a complete single faced function 46

Figure 4.5 The p-OBDDs of different 0-single-faced variables 49

Figure 4.6 p-OBDDs of two different 1-single-faced variables 50

Figure 5.1 n-triangle and 1-triangle 54

Figure 5.2 (6,6) 0-1 grid 55

Figure 5.3 First kind of connection 56

Figure 5.4 Second kind of connection 56

Figure 5.5 The connection of the left-tail function with the connection graph 58

Figure 5.6 The Odd-even parity function 58

Figure 5.7 The reduced OBDD 59

Figure 6.1 Example of an FBDD and an OBDD for the same function f 65

Figure 6.2 FBDD of F_4 in Example 6.1 65

Figure 7.1 Binary Graph Representation of an Integer List 74

Figure 7.2 Another Binary Graph of an Integer List 74

Figure 7.3 Straight Map and Standard Map 78

Figure 7.4 OBDDs and Binary Graphs of D-lists 81

Figure 7.5 Two paths share node v_0 89

List of Tables

Table 5.1 CPU Time..60

Chapter 1 Introduction

In this chapter, the motivation and objective of the research is outlined, together with the overall organization of the thesis.

§1.1 Motivation and Objectives

Boolean functions are a part of the foundation of computer science. Boolean function manipulation is an important component of many algorithms including logic optimization, logic verification of combinational and sequential circuits, and logic synthesis. Many problems in artificial intelligence and combinatorics can also be expressed as a sequence of operations on Boolean functions [1]. Unfortunately, many problems on Boolean functions, such as the satisfiability problem and the tautology checking problem, are NP-complete or co NP-complete problems [32]. Often the degree of difficulty of the problem depends on the size of the Boolean function representation. For functions with large representations, algorithms for tautology checking or satisfiability will take much time.

For n variables, there are 2^{2^n} Boolean functions, which form the function space $F(n)$. The idea of a function is a mathematical concept. To manipulate functions in computers, they must be represented. When n is not small, functions generally have large representations [10], [11], [13]. Therefore much effort has been put into the study of Boolean function representations. This work has had three aspects: (1) finding new data structures; (2) for a specific data structure, finding the representation with minimal cost; and (3) studying the lower and upper bounds of the representation. The first aspect is research on new data structures. The second aspect is studied under the name of

minimization such as two level logic minimization [4], multi-level logic minimization [5], and OBDD minimization. The third aspect is studied in complexity theory.

There are many data structures for Boolean functions. In general, they can be classified as table based, formula based, and graph based. Boolean functions can be represented by truth tables, or equivalently Binary Decision Trees. However the exponential size of truth tables and Binary Decision Trees makes them impractical for functions with many variables. Other traditional formula-based Boolean function representations include two-level logic expressions (SOP and POS forms), multi-level logic expressions (factored forms), and Reed-Muller expansions. It is widely recognized that there are two important criteria for evaluating data structures: (1) compactness, and (2) canonicality. SOP, POS, factored forms, and Reed-Muller expansions fail on both criteria. Truth tables and Binary Decision Trees are canonical, but not compact. Therefore these data structures are not ideal for Boolean functions.

In 1986, a graphic data structure of Boolean functions, Ordered Binary Decision Diagrams (OBDD), was developed by Bryant [1]. Compared with truth tables and Binary Decision Trees, OBDDs are usually much more compact. A wide range of Boolean functions have polynomial size OBDD representations [1]. The reduced OBDD of a Boolean function is unique for any variable ordering. Because of these properties, OBDDs have been applied to many areas including verification, minimization, and testing. Now OBDDs are considered as the state of the art data structure for Boolean function manipulation.

For any of the above data structures except the truth-table, a Boolean function may have more than one representation in the same data structure. For example, a Boolean function may have many different SOP representations. For the OBDD data structure, a different variable ordering may induce a different graph and the size of the reduced OBDD for a Boolean function may depend on the variable ordering. Some functions have linear size OBDDs for one variable ordering and exponential size OBDDs for another variable

ordering [1]. The role of minimization of Boolean functions for a specific data structure is to find the representation with minimal cost to represent a Boolean function. The minimization of Boolean functions has been studied extensively including two-level logic minimization [4] [48], multi-level logic minimization [5], and OBDD minimization [17] [2] [18] [19] [20] [21] [22].

Traditionally, in order to find the absolute minimal representation, Boolean function minimization algorithms search all the possible representations in the given data structure and find the minimal cost one. This is the approach for OBDD minimization [19], for two-level logic minimization, and for multi-level logic minimization. However, due to the large search space, these algorithms are impractical even for modest n. It is very useful to identify some filters to reduce the complexity of the search space and therefore speed up the minimization process. In the literature, the Essential Prime Implicants (EPIs) and connected components [39] are such filters. In two-level logic minimization, the identification of EPIs can greatly reduce the search space. However, the EPI concept does not carry over to data structures other than the SOP (Sum of Product) forms. The connected component concept sometimes fails for SOP minimization. Generally, filters that work for all data structures are not known.

Boolean function minimization can reduce the complexity of the representation for Boolean functions. However, for some functions, the effect of minimization is very limited. For example, many functions, such as the FHS-function [12], integer multiplication, hidden weighted bit function (HWB) [2], or indirect storage access function [12], are proven to have exponential size OBDDs under any variable ordering. Further understanding about the data structure and the functions can be gained from complexity theory point of view. It is proven in the literature that almost all functions have exponential size OBDDs [10], [11], [13]. In order to efficiently represent those hard functions, many new graphic data structures such as FBDD have been proposed, notable are [9], [10], [11], [12], [13], [15], [16], [35], [36], [37], [38].

In this thesis, we study the above three problems, i.e., data structures, minimization, and complexity issues.

Firstly, we re-define the Boolean function minimization problem. The new minimization is defined over the *permutation* equivalent classes of functions instead of single functions. This definition is broader than the traditional Boolean function minimization problem and thereby opens up new avenues for minimization. The OBDD minimization problem is reformulated as well. It is proven that functions in the same permutation equivalent class share the same minimal OBDD. Consequently, any variable ordering will generate equivalent OBDDs for symmetric functions. Therefore OBDD minimization is not needed for symmetric functions.

Secondly, we identify useful filters (heuristics) for Boolean function minimization with respect to any data structures. In the literature, the supporting variable concept is a useful filter for Boolean function minimization with respect to any data structures. The complexity of a function depends on the number of supporting variables. Functions with more supporting variables are more complex, and their minimization is harder. Though the importance of supporting variables has been known for some time [9], no rigorous theory about supporting variables has been formulated. In this thesis, a formal theory about supporting variables and the dimension of Boolean functions is presented. Moreover, supporting variables are further distinguished as *single-faced variables* and *double-faced variables*. It is proven that single-faced variables are a useful filter for Boolean function minimization.

Thirdly, we define a new Boolean function classification theory, called the single-faced function classification. In the literature, many forms of Boolean function classification have been studied. [7] studies the number of equivalent classes under group actions. Some specially identified functions divide $F(n)$ into different classes such as symmetric versus non-symmetric functions and monotonic versus non-monotonic functions. Those classifications are helpful in understanding the complexity and structure of Boolean

functions. They are also helpful in finding the most suitable solutions for each special class of functions and are therefore important in practice [4]. Recently, Boolean function classification has played a role in FPGA architecture design [44]. In this thesis, Boolean functions are classified as single-faced and double-faced functions. Single-faced functions are obtained by single-faced function extensions. Single-faced functions commonly occur. For n variables, there are 2^{2n-1} *complete single-faced* functions, a class of functions with very simple structures. Except for the odd and even parity function, all other double-faced functions contain some single-faced functions as restrictions.

The fourth result is the *complexity of the OBDDs* of complete single-faced functions and symmetric functions. It is proven that complete single-faced functions have linear size OBDDs. For symmetric functions, their OBDD has a lower bound of n^2.

The fifth result is the study of the *structures of OBDDs*. In particular, we study the structure of OBDDs of the single-faced functions and symmetric functions. The structure of symmetric functions consists of triangles and grids as shown in Chapter 5.

Finally, we define some *new data structures* for Boolean functions. These newly defined data structures are dramatically different from all previously proposed data structures, yet they unify all existing data structures as special cases. Some functions are found to have no compact representation in OBDD or other graphic data structures while having compact representation in the new data structure.

§1.2 Overview of the Thesis

This thesis is organized into three parts:

Part I Background.

Chapter 2 We present the basic notations and general concepts. The most important concepts are permutations, Boolean functions, and traditional data structures for Boolean functions. Boolean function minimization is also defined.

Chapter 3 We define OBDDs. Related properties about OBDDs are defined in this chapter. The OBDD minimization problem is reformulated.

Part II Boolean function classification, minimization, and application.

Chapter 4 We present our result on function minimization and the Boolean function classification theory. We also study the structure of the single-faced functions.

Chapter 5 As an application of the single-faced function classification theory, we present our result about symmetric functions. In particular, we present the structure of symmetric OBDDs. An algorithm for symmetry detection is also presented. Experiment results show the advantage of the new algorithm over previous algorithms.

Part III New Data structures.

Chapter 6 We present examples which OBDDs fail to represent efficiently. Other data structures such as FBDD are introduced.

Chapter 7 We further propose some new data structures for Boolean functions based on integer list representation. Moreover, we show the new data structure can unify all known data structures. BDDs and SOP forms are special cases of the new data structure. Some functions are found to have compact representation only in the new data structure.

Part IV Conclusion of the thesis.

Part I Background

In this part, we define notation for permutations, Boolean functions, and their representations. We also define the Boolean function minimization problem.

Chapter 2 Boolean Functions, Their Representations and Minimization

In this chapter, we introduce the basic notation and concepts about permutations, Boolean functions, the traditional data structures for Boolean functions, and the Boolean function minimization problem. The Reduced Ordered SOP form is a new concept developed by the author.

§2.1 Permutation

For any finite set S, $|S|$ denotes the number of elements in S. The character n and m represent natural numbers. The notation $[m, n]$ denotes the set of integers $m \le x \le n$. The conventional notation for sets, such as $\cap$, $\cup$, and $\subset$, are also used in this thesis.

For two finite sets A and B, a *map* $P: A \to B$ is a correspondence such that for any element $a \in A$, $P(a) \subset B$. The set $P(a)$ is the *image of a*. The set A is the *input domain.* The set B is the *output domain*. The set $P(A) = \bigcup_{a \in A} P(a)$ is the *image* of P. The *inverse map* of P is the map $P^{inv}: P(A) \to A$ such that $P^{inv}(x) = \{a \mid P(a) = x\}$. Given a set of maps $S = \{P_i : A \to B \mid i \in I\}$, $S(A) = \bigcup_{i \in I} P_i(A)$.

The map $P: A \to B$ is *onto* if $P(A)=B$. P is a function if for any $a \in A$, $|P(a)|=1$. Otherwise, P is a *relation*. The map P is *one-to-one* if both P and its inverse are functions. A one-to-one and onto map $P: A \to A$ is called a *permutation over A*. The composition of two maps $P_1 : A \to B$ and $P_2 : B \to C$ is defined to be $(P_2 \bullet P_1)(a) = P_2(P_1(a))$.

For a finite variable set $L_n = \{x_1, \cdots, x_n\}$, a one-to-one and onto map $M_{n,n} : L_n \to L_n$ is a *permutation over* L_n. A permutation $M_{n,n}$ over the set $L_n = \{x_1, \cdots, x_n\}$ can be written as

$\begin{pmatrix} x_1 x_2 \cdots x_n \\ x_{i_1} x_{i_2} \cdots x_{i_n} \end{pmatrix}$, where for *1≤j≤n*, $M_{n,n}(x_j) = x_{i_j}$. The set of all such permutations $M_{n,n}$ is denoted by $E_{n,n}$. An *ordering* of L_n is an arrangement of variables such that $x_{i_1} < x_{i_2} < \cdots < x_{i_n}$. The set of all orderings over L_n is denoted by *Or(n)* or $Or(x_1, \cdots, x_n)$. The *standard ordering* is the ordering $x_1 < x_2 < \cdots < x_n$. There is a one-to-one correspondence between the set *Or(n)* and $E_{n,n}$, i.e., every ordering is induced by a permutation. More specifically, the ordering $x_{i_1} < x_{i_2} < \cdots < x_{i_n}$ is induced by the permutation $\begin{pmatrix} x_1 x_2 \cdots x_n \\ x_{i_1} x_{i_2} \cdots x_{i_n} \end{pmatrix}$. Therefore we do not distinguish between orderings and permutations of variables. For an ordering $P = \begin{pmatrix} x_1 x_2 \cdots x_n \\ x_{i_1} x_{i_2} \cdots x_{i_n} \end{pmatrix}$ over L_n and a variable subset $S \subset L_n$, the minimal variable of *S*, denoted by *min(S)*, is $\min(S) = P(x_j) = x_{i_j}$, where $j = \min\{k | P(x_k) \in S\}$.

We define some operations on the set $E_{n,n}$: the inverse permutation and the product of two permutations.

The *inverse permutation* of $M_{n,n}$ is the map $M_{n,n}^{inv}(x_{i_j}) = x_j$, if $M_{n,n}(x_j) = x_{i_j}$.

The *product of two permutations* is a new permutation, denoted as $P_1 * P_2 = P_3$, such that $P_3(x_i) = (P_1 * P_2)(x_i) = P_1(P_2(x_i))$.

$E_{n,n}$ is closed under these two operations and forms the permutation group with *identity* as the permutation $I = \begin{pmatrix} x_1 x_2 \cdots x_i \cdots x_n \\ x_1 x_2 \cdots x_i \cdots x_n \end{pmatrix}$, which maps each element to itself.

We define pairwise permutation of two variables $\{x_i, x_j\}$ to be of the form $\begin{pmatrix} x_1 x_2 \cdots x_i \cdots x_j \cdots x_n \\ x_1 x_2 \cdots x_j \cdots x_i \cdots x_n \end{pmatrix}$. When x_i and x_j are adjacent, i.e., *j=i+1*, the pairwise permutation is called a *basic permutation.*

§2.2 Boolean Functions and Their Representations

Let $B = \{0,1\}$. The n-dimensional Boolean space $B^n = \{(x_1,\cdots,x_n) \mid x_i = 0,1\}$ is the n-fold product of B. A *variable* is a symbol representing a single coordinate of the Boolean space B^n. For the n dimensional Boolean space B^n, there are n variables $\{x_1, x_2, \cdots, x_n\}$ to represent the n coordinates. A *literal* is a variable x_i (positive literal) or its negation $\bar{x}_i$ (negative literal). The positive literal x_i and the negative literal $\bar{x}_i$ are said to be *opposite literals*. We use $\dot{x}_i$ to represent either a positive or negative literal while $\bar{\dot{x}}_i$ is its negation.

A *minterm* is a an instance of B^n. The *distance* between two minterms is the number of coordinates in which the two minterms differ. For example, the minterm (0, 0, 0, 0) and minterm (1, 0, 0, 1) have distance 2. Two minterms are said to be *adjacent* if their distance is 1. The *weight* of a minterm is the number of 1's in the minterm.

A *product term* is a product of literals. A product term can be represented by the literals that form the product such as x_1x_3. If a product contains both a variable and its negation, then it is a *null product*. A product term in B^n can also be represented by the cubical notation, which is a string consisting of n characters. If the product term contains the literal x_i ($\bar{x}_i$), then the *ith* position of the string is 1 (0); otherwise, the *ith* position is represented by "-". For example, in B^3, the product term x_1x_3 is represented as 1-1 in the cubical notation.

The n-dimensional Boolean space B^n is an *n-dimension hypercube*. Each product term is called a *subcube* or a *cube*. A literal x_i represents those minterms whose *i-th* coordinate is 1, its negation $\bar{x}_i$ represents those minterms whose *i-th* coordinate is 0. They are both *n-1* dimensional hypercubes, and are called *a face* of B^n. Moreover, $B^n = x_i \cup \bar{x}_i$. A product term represents the intersection of the minterms represented by each literal in the product term. A null product term does not represent any minterm. The *dimension* of a product term is the dimension of the hypercube it represents. For a non-null product P, its

dimension is n-k, where k is the number of different literals in P. A minterm is a 0 dimensional product term.

Two product terms are said to be *disjoint* if the minterm sets they represent have empty intersection. Two product terms are disjoint if and only if there is at least one variable such that the variable and its negation are contained in the two product terms, respectively. A product term B is said to be *contained* in a product term C, denoted by $B \subset C$, if every literal of C is in B. If for a cube B, $B \subset C$, then every minterm in B is also in C. Two product terms are *adjacent* if, excepting one literal, all literals are the same for the two product terms. For the different literal, the literals in the two product terms are opposite literals.

Definition 2.1 A Boolean function of n variables $\{x_1, \cdots, x_n\}$ is a function $f(x): B^n \to B$, where $x = [x_1, \cdots, x_n]$ is the argument. The complement of function f, denoted by $\bar{f}(x)$, is defined as $\bar{f}(x) = 0 \Leftrightarrow f(x) = 1$ and $\bar{f}(x) = 1 \Leftrightarrow f(x) = 0$.

The set of all Boolean functions $f(x)$: $B^n \to B$ is denoted by $F[x_1, \cdots, x_n]$, or by $F(n)$ if variables are not being considered.

There are many data structures for representing Boolean functions. In general, they can be classified as table based, formula based, and graph based. Boolean functions can be represented by truth tables, or equivalently the Binary Decision Trees. Other traditional formula-based Boolean function representations include two-level logic expressions (SOP and POS forms), multi-level logic expressions (factored forms), and Reed-Muller expansions. Recently there are many graphic data structures have been developed.

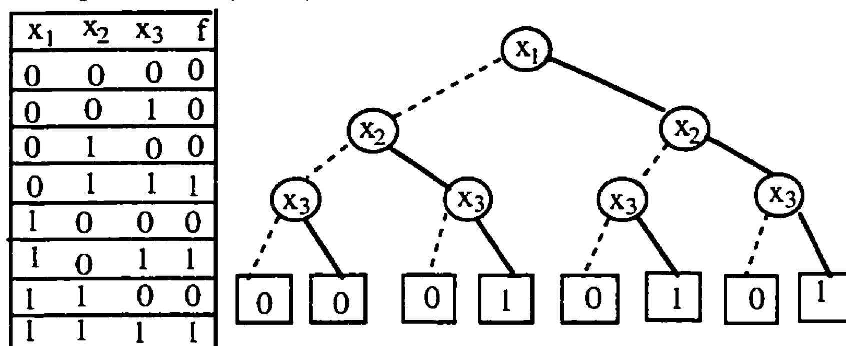
A Boolean function f can be represented by a truth-table which lists all the input minterms and the function values for those minterms. Two Boolean functions which have the same truth-table are identical. Therefore the truth-table is a canonical representation for Boolean functions.

A truth-table can be represented by a complete binary tree. Each vertex in the tree represents a variable. The two children of the vertex represent the cases where the variable has value 0 and 1. This binary tree is called a *decision tree.*

Definition 2.2 ([3]) A *Binary-Decision Tree* (BDT) is a rooted binary tree consisting of two types of vertices: non-terminal vertices and terminal vertices. Each non-terminal vertex v is labeled by a variable *index(v)* and has arcs directed toward two children: *lo(v)* corresponding to the case where the variable is assigned 0 and *hi(v)* corresponding to the case where the variable is assigned to 1. Each terminal vertex is labeled 0 or 1.

Binary Decision Trees can be reduced to form the well-known Ordered Binary Decision Diagram, which will be introduced in the next chapter.

Example 2.1 The following BDT represents the function shown in the truth table, where the left child of vertex v is the *lo(v)*. The terminal value is the function value of the minterm represented by the path reaching the terminal node.



x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Figure 2.1 Truth table and the corresponding complete binary decision tree

Both truth tables and Binary Decision Trees are of exponential size in the number of variables. Therefore they are not efficient representations for Boolean functions. In the literature, many formula based Boolean function representations, which are more efficient than the truth tables and BDTs, have been developed. Those representation mainly focus on the set $f^{inv}(1)$ instead of B^n. The set $f^{inv}(1)$ is called the *on-set* of *f*. Alternatively, we can re-define Boolean functions.

Definition 2.3 A n-variable Boolean function *f* is a set of minterms, i.e, $f \subset B^n$.

For formula based representations, the following concept plays an important role.

Definition 2.4 Given the variable set $L_n = \{x_1, x_2, \cdots, x_n\}$ over B^n $(n>0)$ and a product term $P = \dot{x}_{i_1}\dot{x}_{i_2}\cdots\dot{x}_{i_k}$, where $i_1 < i_2 < \cdots < i_k$, for any function $f: B^n \rightarrow B$, the function restriction of f with respect to P is a function $f_P = f_{\dot{x}_{i_1}\dot{x}_{i_2}\cdots\dot{x}_{i_k}}$ defined as follows.

$$f_P = f_{\dot{x}_{i_1}\dot{x}_{i_2}\cdots\dot{x}_{i_k}} : B^{n-k} \rightarrow B$$

$$f_{\dot{x}_{i_1}\dot{x}_{i_2}\cdots\dot{x}_{i_k}}(y_1,\cdots y_{n-k}) = f(x_1,\cdots,x_{i_1-1},\dot{z}_{i_1},x_{i_1+1},\cdots,x_{i_2-1},\dot{z}_{i_2},x_{i_2+1},\cdots,x_{i_k-1},\dot{z}_{i_k},x_{i_k+1},\cdots,x_n),$$

where $\{y_1,\cdots y_{n-k}\} = \{x_1,\cdots,x_{i_1-1},x_{i_1+1},\cdots,x_{i_2-1},x_{i_2+1},\cdots,x_{i_k-1},x_{i_k+1},\cdots,x_n\}$, $\dot{z}_{i_j} = 0$ if $\dot{x}_{i_j} = \bar{x}_{i_j}$; $\dot{z}_{i_j} = 1$ if $\dot{x}_{i_j} = x_{i_j}$.

The above definition has been defined in different forms in the literature and is called *cofactor* and *restriction* [5].

In terms of minterm sets, $f_P = f_{\dot{x}_{i_1}\dot{x}_{i_2}\cdots\dot{x}_{i_k}} = f \cap P$, i.e., the set of minterms in the intersection of the on-set of f and P. A product term P is an *implicant* of a function f if $f_P = f \cap P = 1$, i.e., $P \subset f$. For two product terms P, Q and a Boolean function f, $f_{PQ} = (f_P)_Q$.

When the product term is a single literal x_i or $\bar{x}_i$, we have f_{x_i} ($f_{\bar{x}_i}$) denoting the positive (negative) cofactor of f with respect to variable x_i, i.e., the function resulting when constant 1, (0) is substituted for x_i. For any function f, $f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}$.

Definition 2.5 For an n $(n>0)$ dimensional Boolean space B^n with variable set $L_n = \{x_1,\cdots,x_n\}$, a variable $x_i \in L_n$ is a *supporting variable* of $f \in F(n)$ if $f_{x_i} \neq f_{\bar{x}_i}$. For a literal x_i in an implicant $P = x_1 x_2 \cdots x_k$ of function f, x_i is a redundant literal of P if $x_i \notin \sup(f_{x_1\cdots x_{i-1}x_{i+1}\cdots x_k})$. If an implicant of a function does not contain any redundant literal, then this implicant is a *prime implicant (PI)*.

This definition also holds for product term $P = x_{i_1}x_{i_2}\cdots x_{i_k}$. The same convention holds for the next few definitions where we only define the case $P = x_1 x_2 \cdots x_k$.

Proposition 2.1 A literal x_i of an implicant $P = x_1 x_2 \cdots x_i \cdots x_k$ of f is redundant iff $P_1 = x_1 x_2 \cdots \bar{x}_i \cdots x_k$ is an implicant of f. In this case, $P_2 = x_1 x_2 \cdots x_{i-1} x_{i+1} \cdots x_k$ ($P_2 \supset P$), is

also an implicant of *f*. For a prime implicant *A*, there is no other implicant of *f* that contains A.

Proof If x_i is redundant in $P = x_1 x_2 \cdots x_i \cdots x_k$, then $x_i \notin \sup(f_{x_1 \cdots x_{i-1} x_{i+1} \cdots x_k})$. Therefore $f_{x_1 \cdots x_{i-1} x_i x_{i+1} \cdots x_k} = f_{x_1 \cdots x_{i-1} \bar{x}_i x_{i+1} \cdots x_k} = 1$, i.e., $P_1 = x_1 x_2 \cdots \bar{x}_i \cdots x_k$ is an implicant of *f*. The converse is also true. Therefore we can see the conclusion.

These prime implicants can be generalized to the case in which variable sets are ordered. In the following definition, *min(sup(f))* denotes the minimum element of the ordered variable set *sup(f)*. Without losing generality, we assume the ordering is the standard ordering.

Definition 2.6 For an ordered variable set $L_n = \{x_1, \cdots, x_n\}$, an implicant $P = x_1 x_2 \cdots x_k$ of function *f* over L_n is said to be *ordered implicant (OI)* of *f* if for $1 < i \le k$, $x_i \le \min(\sup(f_{x_1 \cdots x_{i-1}}))$, $x_1 \le \min(\sup(f))$. A literal x_i in an OI $P = x_1 x_2 \cdots x_k$ of f is a *redundant literal* if $x_i < \min(\sup(f_{x_1 \cdots x_{i-1}}))$. If an OI does not contain any redundant literal, then it is an *ordered prime implicant (OPI)* of *f*.

If a literal x_i is a redundant literal in an OI $P = x_1 x_2 \cdots x_k$ of function *f* over L_n, then $x_i < \min(\sup(f_{x_1 \cdots x_{i-1}}))$, which implies that $x_i \notin \sup(f_{x_1 \cdots x_{i-1}})$. Therefore redundant literals in OIs are redundant in the normal sense.

By definition, a product term $P = x_1 x_2 \cdots x_k$ is an OPI of a function f iff for all $1 < i \le k$, $x_i = \min(\sup(f_{x_1 \cdots x_{i-1}}))$, $x_1 = \min(\sup(f))$, therefore an OPI may not be a PI of the function. The literal x_i in an OPI is determined by the supporting set of $f_{x_1 \cdots x_{i-1}}$. It has to be $\min(\sup(f_{x_1 \cdots x_{i-1}}))$. However it is possible that x_i is a redundant literal in normal sense, i.e., $x_i \notin \sup(f_{x_1 \cdots x_{i-1} x_{i+1} \cdots x_k})$.

Example 2.2 Suppose a function is given by the minterm set of $f = x_1 x_2 + x_3 x_4$. The ordering is the standard ordering $x_1 < x_2 < x_3 < x_4$. The product term : $x_1 \bar{x}_2 x_3 x_4$. is an OPI of *f* since $x_1 = \min(f)$, $\bar{x}_2 = \min(f_{x_1})$, $x_3 = \min(f_{x_1 \bar{x}_2})$, $x_4 = \min(f_{x_1 \bar{x}_2 x_3})$, while $x_1 \bar{x}_2 x_3 x_4$ is not a prime implicant of *f*.

Corollary 2.1 If $P=$ $x_1x_2\cdots x_k$ is an OPI of f, then $x_i\cdots x_k$ is an OPI of function $f_{x_1\cdots x_{i-1}}$

Proposition 2.2 For an OI P of f, the following two results are true.

(1) If P is an OPI, then there no other OI of f can contain P.

(2) Conversely, if no other OI of f contains P, P is an OPI.

Proof (1) If $P=$ $x_1x_2\cdots x_k$ is an OPI, then for any $1\le i\le k$, $x_i=\min(\sup(f_{x_1\cdots x_{i-1}}))$, and $x_{i+1}>x_i$. If there is another product term $P_1=x_1\cdots x_{i-1}x_{i+1}\cdots x_k$ is an OI of f, then $x_{i+1}\le\min(\sup(f_{x_1\cdots x_{i-1}}))=x_i$, we have a contradiction. Therefore the result is correct.

(2) This result is a direct conclusion of the following lemma.

Lemma If a literal x_i of an OI $P=x_1x_2\cdots x_i\cdots x_k$ is redundant, then $P_2=x_1x_2\cdots x_{i-1}\bar{x}_i x_{i+1}\cdots x_k$ is an OI, so is $P_3=x_1x_2\cdots x_{i-1}x_{i+1}\cdots x_k$.

Proof If a literal x_i of an OI $P=x_1x_2\cdots x_i\cdots x_k$ is redundant, $x_i\notin\sup(f_{x_1\cdots x_{i-1}})$, then $f_{x_1\cdots x_{i-1}x_i}=f_{x_1\cdots x_{i-1}\bar{x}_i}$, therefore $1=f_{x_1\cdots x_{i-1}x_ix_{i+1}\cdots x_k}=f_{x_1\cdots x_{i-1}\bar{x}_ix_{i+1}\cdots x_k}$, which implies that $x_1\cdots x_{i-1}\bar{x}_ix_{i+1}\cdots x_k$ is also an implicant of f. Moreover, for all $1\le j\le k$, $j\ne i$, $x_j\le\min(\sup(f_{x_1\cdots x_{j-1}}))$, $\bar{x}_i<\min(\sup(f_{x_1\cdots x_{i-1}}))$. Therefore $P_2=x_1x_2\cdots x_{i-1}\bar{x}_ix_{i+1}\cdots x_k$ is an OI. Since both P and P_2 are implicants of f, $P+P_2$ = $P_3=x_1x_2\cdots x_{i-1}x_{i+1}\cdots x_k$ is an implicant of f. For all $1\le j<i$, $x_j\le\min(\sup(f_{x_1\cdots x_{j-1}}))$. For $i<j\le k$, $\sup(f_{x_1\cdots x_i\cdots x_{j-1}})$ = $\sup(f_{x_1\cdots x_{i-1}x_{i+1}\cdots x_{j-1}x_i})$ = $\sup(f_{x_1\cdots x_{i-1}x_{i+1}\cdots x_{j-1}})$, therefore, $x_j\le\min(\sup(f_{x_1\cdots x_i\cdots x_{j-1}}))$= $\min(\sup(f_{x_1\cdots x_{i-1}x_{i+1}\cdots x_{j-1}}))$. So P_3 is an OI.

Corollary 2.2 Two different OPIs are disjoint.

Proof This can be proven by induction on the number of literals in the OPIs. Suppose the OPIs both have only one literal, then they have to be opposite literals and therefore disjoint. If one of them has one literal and another one has more than one literals, then they have to be disjoint as well since the first literals of them, which are the literals of the minimum variable in the supporting set of the function, have to be different. Otherwise they could not be both OPIs by the above proposition. Now for general case, suppose $P_1=x_1x_2\cdots x_{k_1}$ and $P_2=y_1y_2\cdots y_{k_2}$ are two OPIs, then either $x_1=y_1$ or $\bar{x}_1=y_1$. If $\bar{x}_1=y_1$,

then the conclusion is right. If $x_1 = y_1$, then $y_2 \cdots y_{k_2}$ and $x_2 \cdots x_{k_1}$ are two OPIs of the function f_{x_1}, they must be disjoint by induction. Therefore the conclusion is right.

Corollary 2.3 For any OI, there is one and only one OPI that contains this OI.

Proof Because two OPIs are disjoint, there are no two OPIs that contains this OI. On the other hand, if this OI is not an OPI, then there is another OI contains it. Therefore it is contained in an OPI.

This OPI can be found by deleting all the literals $x_i < \min(\sup(f_{x_1 \cdots x_{i-1}}))$.

Implicants, ordered implicants, prime implicants, and ordered prime implicants can be used as the building blocks for the formula based Boolean function representation.

Definition 2.7 A Boolean function is in SOP form if it is a sum of implicants.

A Boolean function can be represented by many different SOP forms.

Definition 2.8 Given an ordered variable set $L_n = \{x_1, \cdots, x_n\}$ and a SOP form f over L_n, f is an *ordered SOP form (OSOP)* if every product term in f is an OI of f. An ordered SOP form f is *irredundant* if there are no two product terms C and D in f such that $C \supset D$. If an OSOP form has every product term to be an OPI, then the SOP form is called a *reduced ordered SOP form (ROSOP).*

It is easy to see that a ROSOP form has any two product terms disjoint from each other.

Proposition 2.3 With respect to any variable ordering, the ROSOP representation of any Boolean function consists of all the OPIs and only all OPIs of the function. Therefore the ROSOP form is unique for any function with respect to a variable ordering.

Proof It is obvious that two different Boolean functions have different ROSOP forms. On the other hand, every Boolean function has only one ROSOP representation, because every minterm of the function is an OI, and every OI is contained in only one OPI.

Example 2.3 Suppose the function is given by the minterm set of $f = x_1x_2 + x_3x_4 + x_5x_6$. The ordering is the standard ordering $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$, Using the algorithm described in [33], we can derive the OPIs as $x_1x_2, x_1\bar{x}_2x_3x_4$, $x_1\bar{x}_2x_3\bar{x}_4x_5x_6, x_1\bar{x}_2\bar{x}_3x_5x_6, \bar{x}_1x_3x_4, \bar{x}_1x_3\bar{x}_4x_5x_6, \bar{x}_1\bar{x}_3x_5x_6$: Therefore in terms of OPIs,

$f = x_1x_2 + x_1\bar{x}_2x_3x_4 + x_1\bar{x}_2x_3\bar{x}_4x_5x_6 + x_1\bar{x}_2\bar{x}_3x_5x_6 + \bar{x}_1x_3x_4 + \bar{x}_1x_3\bar{x}_4x_5x_6 + \bar{x}_1\bar{x}_3x_5x_6$.

A SOP form is a special factored form. General factored forms are defined as follows.

Definition 2.9 A *factored form f* of Boolean function is one of the following: (1) a constant 1 or 0, (2) a literal, (3) $\sum_{i \in I} f_i$, or (4) $\prod_{i \in I} f_i$, for $i \in I$, f_i are factored forms.

The factored form representation of a Boolean function has some attractive properties. A factored form represents both a function and its complement by duality [29]. There is a tree structure isomorphic to the factored form. Each internal node of the tree is an AND or OR operator and each leaf is a literal or constant. Some special factored forms play important roles in the data structure of Boolean functions. One of them is the function expression, $f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}$, which is called the Shannon expansion of function *f*. More formally we have the following definition.

Definition 2.10 Given a variable set $L_n = \{x_1, \cdots, x_n\}$, a Shannon expansion f is defined to be of the form: (1) constant 1 or 0, or (2) $f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}$, where both f_{x_i} and $f_{\bar{x}_i}$ are Shannon expansions.

Example 2.4 From the ROSOP form $f = x_1x_2 + x_1\bar{x}_2x_3x_4 + x_1\bar{x}_2x_3\bar{x}_4x_5x_6 + x_1\bar{x}_2\bar{x}_3x_5x_6 + \bar{x}_1x_3x_4 + \bar{x}_1x_3\bar{x}_4x_5x_6 + \bar{x}_1\bar{x}_3x_5x_6$, we can arrive at the Shannon expansion as follows, using algebraic factorization:

$f = x_1\{x_2 + \bar{x}_2[x_3(x_4 + \bar{x}_4x_5x_6) + \bar{x}_3x_5x_6]\} + \bar{x}_1[x_3(x_4 + \bar{x}_4x_5x_6) + \bar{x}_3x_5x_6]$..

Shannon expansions play an important role in Boolean function manipulation. The well-known logic minimization algorithm ESPRESSO in [4] is based on Shannon expansions. Shannon expansions are also the basis for the OBDD and other Decision Diagram data structures such as FBDDs.

One more formula based Boolean function representation is the Reed-Muller expansion [46]..

Definition 2.11 A Reed-Muller expansion is defined to be an EX-OR sum of product terms.

Example 2.5 The odd parity function can be represented by $x_1 \oplus x_2 \oplus \cdots \oplus x_n$. This function evaluates to 0 if there is even number of variables equal to 1, and evaluates to 1 if there is odd number of variables equal to 1.

Reed-Muller expansions and SOP forms have their own strengths and weaknesses. For example, the above odd and even parity function need exponential number of product terms if the SOP form is used. However, it is linear in the Reed-Muller expansion.

There are some common problems with the various representations of Boolean functions we have introduced. First, certain common functions require representations of exponential size. For example, the even and odd parity functions serve as worst case examples in SOP and factored forms. Second, while a certain function may have a reasonable representation, performing a simple operation such as complementation could yield a function with an exponential representation. Finally, none of these representations (except ROSOP forms) are canonical forms, i.e., a given function may have many different representations. Consequently, testing for equivalence or satisfiability is difficult [1]. Due to these characteristics, most programs that process a sequence of operations on Boolean functions have rather erratic behavior. They proceed at a reasonable pace, but then suddenly "blow up", either running out of storage or failing to complete an operation in a reasonable amount of time. To overcome those problems, a new data structure called Ordered Binary Decision Diagrams (OBDD) was introduced. We review the OBDDs in the next chapter.

For all those representation forms except truth-tables, we have the following problem.

Representation Minimization Problem I Given a Boolean function data structure, for any function $f \in F(n)$, find the minimal cost representation of *f*.

In the literature, the Representation Minimization Problem I has been intensively studied under different names such as two-level logic (SOP) minimization and multi-level logic minimization. SOP minimization is the tradition logic minimization problem, for which some well-known solutions exist [4]. The multi-level logic minimization problem is

also an active research area [5]. The OBDD minimization problem is also defined for single functions in the literature.

§2.3 Permutations on Boolean Functions

Permutations on the variable set of Boolean spaces induce permutations in the Boolean spaces. They also induce permutations over the function space *F(n)*.

Definition 2.12 For a permutation of n variables $M_{n,n} = \begin{pmatrix} x_1 x_2 \cdots x_n \\ x_{i_1} x_{i_2} \cdots x_{i_n} \end{pmatrix}$ over the variable set $L_n = \{x_1, \cdots, x_n\}$, the *induced Boolean space permutation* is defined as: for any minterm $\dot{x}_1 \dot{x}_2 \cdots \dot{x}_n$, $M_{n,n}(\dot{x}_1 \dot{x}_2 \cdots \dot{x}_n) = \dot{x}_{i_1} \dot{x}_{i_2} \cdots \dot{x}_{i_n}$.

This induced map over Boolean spaces is also denoted by $M_{n,n}$.

Definition 2.13 For a permutation $M_{n,n} = \begin{pmatrix} x_1 x_2 \cdots x_n \\ x_{i_1} x_{i_2} \cdots x_{i_n} \end{pmatrix}$ and a function $f \in F(n)$, the function $M_{n,n}(f)$ is defined as $M_{n,n}(f)(x_1, \cdots, x_n) = f(M_{n,n}(x_1, \cdots, x_n)) = f(x_{i_1}, \cdots, x_{i_n})$. This function is called a permutation of function *f*. Two functions $f_1 \in F(n)$ and $f_2 \in F(n)$ are *permutation equivalent* if there is a permutation $M_{n,n} \in E_{n,n}$ such that $f_1 = M_{n,n}(f_2)$. For a function $f \in F(n)$, the set of all functions which are permutation equivalent to f is denoted by $E_{n,n}(f)$, which is an equivalence class of *f*. A function is *symmetric* if $E_{n,n}(f)$ has only one element. The set of all equivalence classes of functions in *F(n)* is denoted by $F(n)/E_{n,n}$.

Example 2.6 $f(x_1, x_2, x_3) = x_1 + x_2 \bar{x}_3$ is equivalent to function $g(x_1, x_2, x_3) = x_2 + \bar{x}_1 x_3$ under the variable permutation $M_{3,3}(x_1, x_2, x_3) = (x_2, x_3, x_1)$.

The set of permutation equivalent classes of functions $F(n)/E_{n,n}$ is very important. In particular, Representation Minimization Problem I can be extended into the following problem.

Representation Minimization Problem II Given a Boolean function data structure, for a function class $E_{n,n}(f) \in F(n)/E_{n,n}$, find the minimal cost representation of $E_{n,n}(f)$.

Representation Minimization Problem II defined here is broader than the minimization encountered in the literature, which is usually Representation Minimization Problem I. In this thesis, we will address this problem with the OBDD data structure. It will be shown that the OBDD minimization problem is indeed the Representation Minimization Problem II for the OBDD data structure.

Chapter 3 OBDD Review

In this chapter, we introduce OBDD related concepts. We also introduce new concepts about OBDD permutations. The OBDD minimization problem is re-formulated.

§3.1 OBDDs

In Chapter 2, we reviewed the traditional data structures for Boolean functions and pointed out their limitations. Those limitations are the driving force for people seeking new data structures. Among them, Ordered Binary Decision Diagrams (OBDDs) are considered to be the state of the art in data structures for Boolean functions. In practice, OBDDs are found to be more compact for Boolean function representation. Examples in the later section will show that some functions require exponential size SOP representations while requiring only linear size OBDDs. OBDDs can be considered as the graphic representation of Shannon expansions with sharing of common subexpressions. It can also be viewed as being obtained from BDT by removing some duplicated subtrees. The formal definition of OBDD is given below.

Definition 3.1 Given an ordered variable set $L_n = \{x_1, \cdots, x_n\}$, an Ordered Binary Decision Diagram (OBDD) is a rooted directed graph with vertex set V containing two types of vertices. A non-terminal vertex v has as attribute an argument index *index(v)*$\in$ $L_n = \{x_1, \cdots, x_n\}$ and two children *lo(v), hi(v). A terminal vertex v* has as attribute a value $value(v) \in \{0,1\}$.

Furthermore, under the ordering < over $L_n = \{x_1, \cdots, x_n\}$, we require that for any non-terminal vertex v, if *lo(v)* is also non-terminal, then we must have *index(v)< index(lo(v))*. Similarly, if *hi(v)* is non-terminal, then we must have *index(v)<index(hi(v))* .

The function represented by an OBDD follows the following rule, which will give us the Shannon expansion of the Boolean function represented by the OBDD.

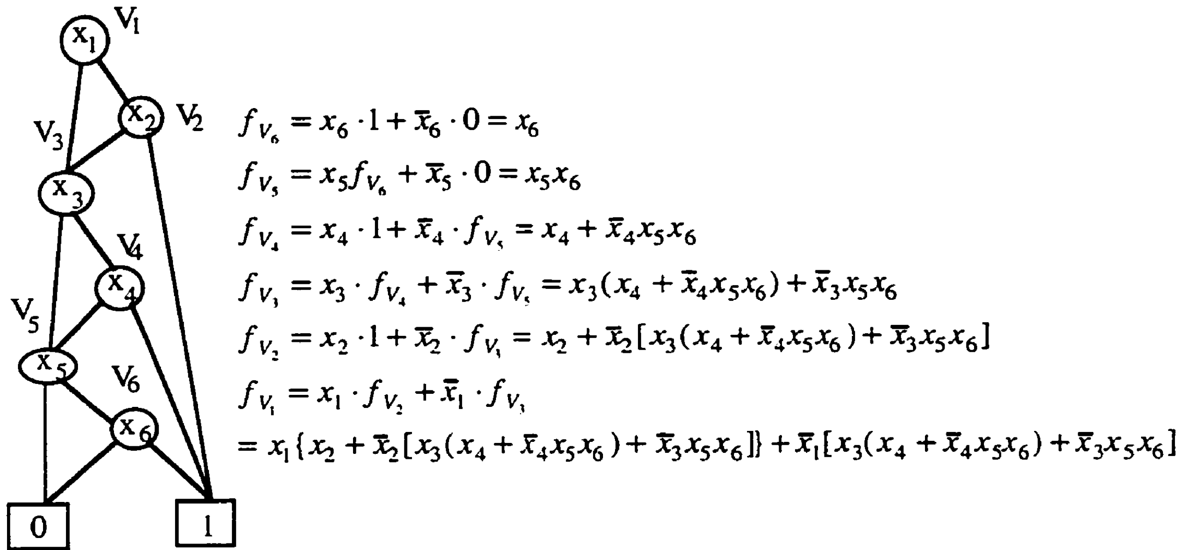
Definition 3.2 An OBDD with root vertex v denotes a function f_v defined recursively as:

(1) If v is a terminal vertex, then f_v=*value(v)*.

(2) If v is a non-terminal vertex with *index(v)*= x_i, then f_v is the function

$$f_v(x_1,\cdots,x_n) = \bar{x}_i f_{lo(v)}(x_1,\cdots,x_n) + x_i f_{hi(v)}(x_1,\cdots,x_n).$$

Example 3.1 The OBDD in the following figure represents the function f_{V_1}, where the ordering is the standard ordering $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$. For a vertex V_i, the right child is $hi(V_i)$, while the left child is *lo(* V_i*)*.



$$f_{V_6} = x_6 \cdot 1 + \bar{x}_6 \cdot 0 = x_6$$
$$f_{V_5} = x_5 f_{V_6} + \bar{x}_5 \cdot 0 = x_5 x_6$$
$$f_{V_4} = x_4 \cdot 1 + \bar{x}_4 \cdot f_{V_5} = x_4 + \bar{x}_4 x_5 x_6$$
$$f_{V_3} = x_3 \cdot f_{V_4} + \bar{x}_3 \cdot f_{V_5} = x_3(x_4 + \bar{x}_4 x_5 x_6) + \bar{x}_3 x_5 x_6$$
$$f_{V_2} = x_2 \cdot 1 + \bar{x}_2 \cdot f_{V_3} = x_2 + \bar{x}_2[x_3(x_4 + \bar{x}_4 x_5 x_6) + \bar{x}_3 x_5 x_6]$$
$$f_{V_1} = x_1 \cdot f_{V_2} + \bar{x}_1 \cdot f_{V_3}$$
$$= x_1\{x_2 + \bar{x}_2[x_3(x_4 + \bar{x}_4 x_5 x_6) + \bar{x}_3 x_5 x_6]\} + \bar{x}_1[x_3(x_4 + \bar{x}_4 x_5 x_6) + \bar{x}_3 x_5 x_6]$$

Figure 3.1 OBDD of $f_{V_1} = x_1x_2 + x_3x_4 + x_5x_6$

Expanding f_{V_1} into SOP form, we have

$$f_{V_1} = x_1x_2 + x_1\bar{x}_2x_3x_4 + x_1\bar{x}_2x_3\bar{x}_4x_5x_6 + x_1\bar{x}_2\bar{x}_3x_5x_6 + \bar{x}_1x_3x_4 + \bar{x}_1x_3\bar{x}_4x_5x_6 + \bar{x}_1\bar{x}_3x_5x_6$$

which is exactly the ROSOP form of $f_{V_1} = x_1x_2 + x_3x_4 + x_5x_6$.

The above result is not a coincidence. Using the following concept, we can directly arrive at the SOP form of the Boolean function represented by the OBDD.

Definition 3.3 A *path* in an OBDD is a sequence of connected vertices starting from the root of the OBDD and ending with a terminal vertex. The literals associated with all

vertices in the path form a product term. This product term is the *associated product term of the path.* The *right-most* path of an OBDD is the path in which all edges are of the form *(v, hi(v))*.

The function represented by an OBDD is the sum of the product terms associated with all paths ending with the 1 terminal vertex.

Example 3.2 For the function $f_{V_1} = x_1x_2 + x_3x_4 + x_5x_6$ and its OBDD shown in previous example, all the associated product terms of the paths ended with the 1 terminal vertex are as follows $x_1x_2, x_1\bar{x}_2x_3x_4, x_1\bar{x}_2x_3\bar{x}_4x_5x_6, x_1\bar{x}_2\bar{x}_3x_5x_6, \bar{x}_1x_3x_4, \bar{x}_1x_3\bar{x}_4x_5x_6, \bar{x}_1\bar{x}_3x_5x_6.$

Definition 3.4 Two OBDDs G and G' are *isomorphic*, denoted by $G=G'$, if there exists a one-to-one function σ from the vertices of G onto the vertices of G' such that for any vertex v if $\sigma(v)=v'$, then either both v and v' are terminal vertices with $value(v)=value(v')$, or both v and v' are non-terminal vertices with $\sigma(lo(v))=lo(v')$, $\sigma(hi(v))=hi(v')$, and $index(v) =index(v')$.

In other words, two OBDDs are isomorphic if their roots have the same variable attributes, and the corresponding children are isomorphic as well. Two terminal vertices that have the same value are isomorphic.

Definition 3.5 An OBDD G is reduced if it contains no vertex v with $lo(v)=hi(v)$, and it does not contain distinct vertices v and v' such that the subgraphs rooted by v and v' are isomorphic.

Theorem 1 ([1]) Given a variable ordering, for any Boolean function f, there is a unique (up to isomorphism) reduced OBDD denoting f. Any other OBDD denoting f contains more vertices.

This proposition is one of the key properties that make OBDDs widely useful. OBDDs are more compact than truth tables. This compact structure is canonical. Therefore verification can be more easily done.

Proposition 3.1 For a reduced OBDD, every path ending with the 1 terminal vertex represents an OPI of the function.

Proof of this proposition can be found in [33], where it is shown how to obtain the ROSOP form of a function from its truth-table. By factoring the ROSOP form, we can obtain the Shannon expansion of the function. Furthermore, Shannon expansions can be represented by OBDDs. Since ROSOP is a unique representation for a function, therefore it serves as a new proof for the canonicality of the OBDDs, which is different from the proof in [1].

Definition 3.6 For a given variable ordering P, a Boolean function f of n variables has a unique reduced OBDD. The reduced OBDD of a function f is denoted by $OBDD_{n,P}(f)$. If the ordering is the standard ordering, then the notation can be simplified to $OBDD_n(f)$. In the rest of the thesis, OBDD refers to the reduced OBDD unless otherwise stated. For an OBDD of the form $OBDD_{n,P}(f)$, $|OBDD_{n,P}(f)|$ denotes the number of vertices in the graph.

§3.2 Partial-OBDDs

In this thesis, we study the structure of OBDDs for some specific functions. In order to do so, we simply the OBDD structures. OBDDs can be reduced to partial-OBDDs [33]. In simple terms, a partial-OBDD is a graph obtained from an OBDD by deleting all the 0-terminal vertices and edges connected to 0 terminal vertices. After some edges and terminal vertices are deleted in an OBDD, some vertices in the graph may have only one child. In order to represent this one child, vertex labeling is not enough. Therefore, partial-OBDDs have labeled edges instead of labeled vertices. Formally, we have the following definition.

Definition 3.7 Given a variable set $L_n = \{x_1, \cdots x_n\}$, a *partial-OBDD* is a rooted directed graph with two types of vertices: non-terminal vertices and terminal vertices. All terminal vertices have value 1. Each non-terminal vertex can have one child or two children. Each edge $e = \{v_1, v_2\}$ in the graph has an $index(e) \in \{x_1, \cdots x_n, \bar{x}_1, \cdots \bar{x}_n\}$. If a vertex has two outgoing edges, then the two edges have opposite literals as indexes.

Furthermore, we impose a total ordering < over the set of variables (both the positive and negative literal of the same variable are of the same order) and require that for any three adjacent vertices $\{v_1, v_2, v_3\}$ connected by edges $\{v_1, v_2\}$ and $\{v_2, v_3\}$, $index(\{v_1, v_2\}) < index(\{v_2, v_3\})$.

The function represented by a partial-OBDD follows similar rules as the representation rule for OBDDs. Moreover, we have the path concept for partial-OBDDs as well.

Definition 3.8 A *path* in an partial-OBDD is defined to be a sequence of adjacent vertices $\{v_1, \cdots, v_k\}$ from the root of the graph to the terminal vertex. The literals associated with all edges in the path form a product term. This product term is the *associated product term of the path.*

The Boolean function represented by a partial-OBDD is the sum of the product terms associated with all paths.

Example 3.3 The OBDD and the partial-OBDD of function $x_1x_2 + x_3x_4 + x_5x_6$ are shown in Figure 3.2.

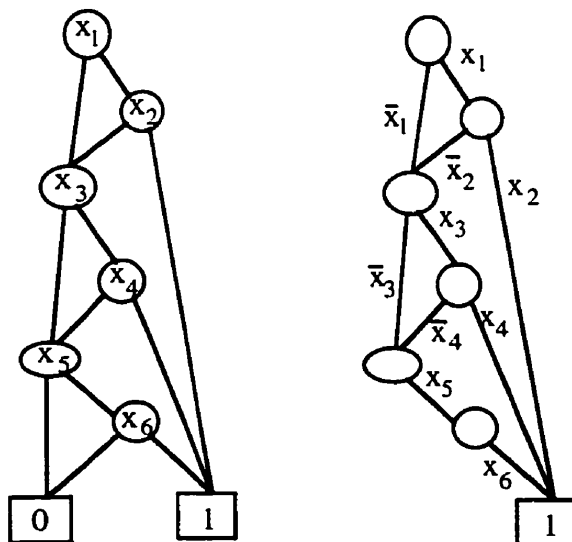


Figure 3.2 The OBDD and partial-OBDD of $x_1x_2 + x_3x_4 + x_5x_6$

Definition 3.9 Two partial-OBDDs are *isomorphic* if their roots have the same number of children, the edges to the children are labeled by the same literals, and the

corresponding children are isomorphic. If the two graphs contain only terminal vertices, then they are identical. We denote two isomorphic partial-OBDDs G_1 and G_2 as $G_1 = G_2$.

Definition 3.10 A partial-OBDD is reduced iff there is no vertex in the graph such that the two children of the vertex are isomorphic, and the graph does not contain distinct vertices v and v' such that the subgraphs rooted by v and v' are isomorphic.

Similar to OBDDs, partial-OBDDs are canonical representations for Boolean functions. From now on, the term partial-OBDDs refers to the reduced partial-OBDDs, unless stated otherwise. For a reduced partial-OBDD, each associated product term of a path is an OPI of the function [33].

Definition 3.11 A non-terminal vertex v of a partial-OBDD is *unate* if v has only one child. Vertices are *binary* if they are not unate. If a vertex v is a unate vertex, the edge starting from this vertex is a *unate edge*. Edges are *binary edges* if they are not unate edges.

Definition 3.12 A *layer* of a (partial-) OBDD is the set of edges labeled by literals of the same variable. Layers are ranked by the ordering of the variables. The *root graph* of an (partial-) OBDD of a variable x is the graph consisting of layers less than or equal to the variable x.

Vertices in the root graph are called *terminal vertices* if they have no child in the root graph. Among them, those which are not terminal vertices of the original partial-OBDD are called *N-type* terminal vertices. A vertex is a *semi-terminal vertex* if it is connected to a terminal vertex. A vertex is an *N-type semi-terminal vertex* if it is connected to a N-type terminal vertex. For a terminal vertex v, the set of all semi-terminal vertices connected to v is denoted by *semi(v)*. Edges in the root graph are called *terminal edges* if they have one terminal vertex as one vertex. Among them, those which have one N-type terminal vertex are *N-type terminal edges*.

§3.3 Permutation and Minimization of OBDDs

For OBDDs, the Boolean function representation minimization problem is the OBDD minimization problem. Since the reduced OBDD has the least number of vertices, the object of minimization of OBDDs is not to find the reduced OBDD with respect to a given variable ordering. Rather the problem is to find the best variable ordering for the function.

Definition 3.13 For a Boolean function *f*, a variable ordering $M_{n,n} \in E_{n,n}$ is called the *minimal ordering* if the condition $|OBDD_{n,M_{n,n}}(f)| = \min_{P \in E_{n,n}} \{|OBDD_{n,P}(f)|\}$ is satisfied.

The size of the reduced OBDD for a Boolean function may vary greatly with the variable ordering. Some functions have linear size OBDDs for one variable ordering and exponential size OBDDs for another variable ordering. To see the importance of OBDD minimization, let us first look at some examples.

Example 3.4 In the following figure, we show the reduced OBDD of the function $x_1x_2 + x_3x_4 + x_5x_6$ under two different variable orderings. The first ordering is the standard ordering $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$, the second ordering is the ordering $x_1 < x_3 < x_5 < x_2 < x_4 < x_6$.

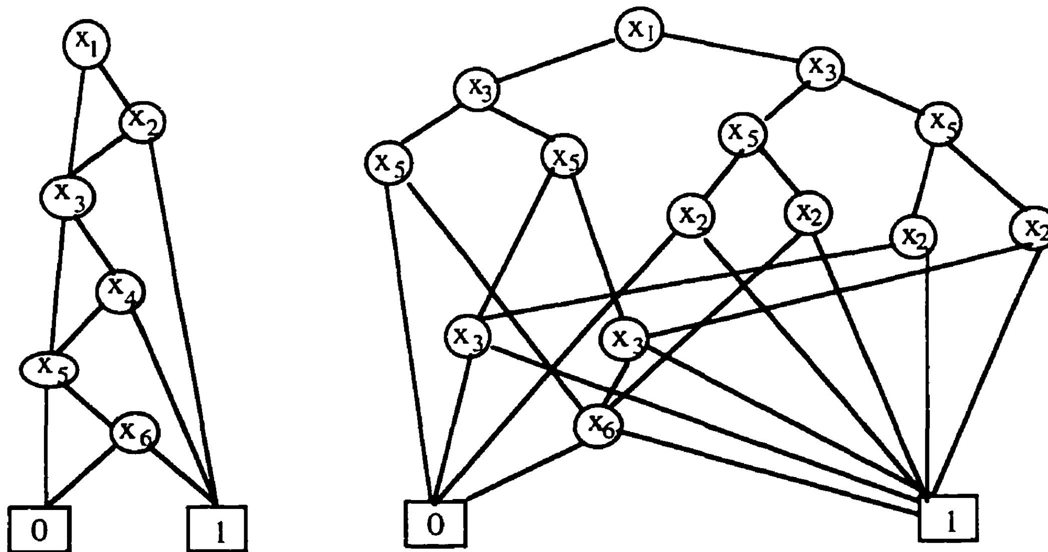


Figure 3.3 The OBDD of $x_1x_2 + x_3x_4 + x_5x_6$ under different variable orderings.

The first graph has only 8 vertices while the second requires 16 vertices. Generalizing this to functions of $2n$ arguments, the function $x_1x_2+\cdots+x_{2n-1}x_{2n}$ is denoted by a graph of $2n+2$ vertices $x_1<x_2\cdots<x_{2n-1}<x_{2n}$, while the ordering $x_1<x_{n+1}<x_2<x_{n+2}<\cdots<x_n<x_{2n}$ requires 2^{n+1} vertices. Consequently, a poor initial choice of input ordering can have very undesirable effects [1].

Example 3.5 In this example, we represent a minterm by its corresponding integer value. The function *F(k)* is a Boolean function of *2k* variables, which is defined by the on-set consisting of 2^k minterms *F(k)*= $\{\sum_{i=0}^{k-1} x_i*2^{i+k}+\sum_{i=0}^{k-1} x_i*2^{k-1-i} | x_i=0,1\}$. In other words, the integers in the minterm set of *F(k)* are of the form $x_{k-1}\cdots x_0x_0\cdots x_{k-1}$. This example also serves an example function whose SOP form is of exponential size while the optimal OBDD is of linear size, therefore it also shows the advantage of OBDDs over SOP forms.

For *k=3*, the function *F(3)={000000, 001100, 010010, 0111110, 100001, 101101, 110011, 111111}*. There are $2^3=8$ minterms in this function. The partial-OBDD of function *F(3)* under the standard ordering is shown in the following Figure 3.4.

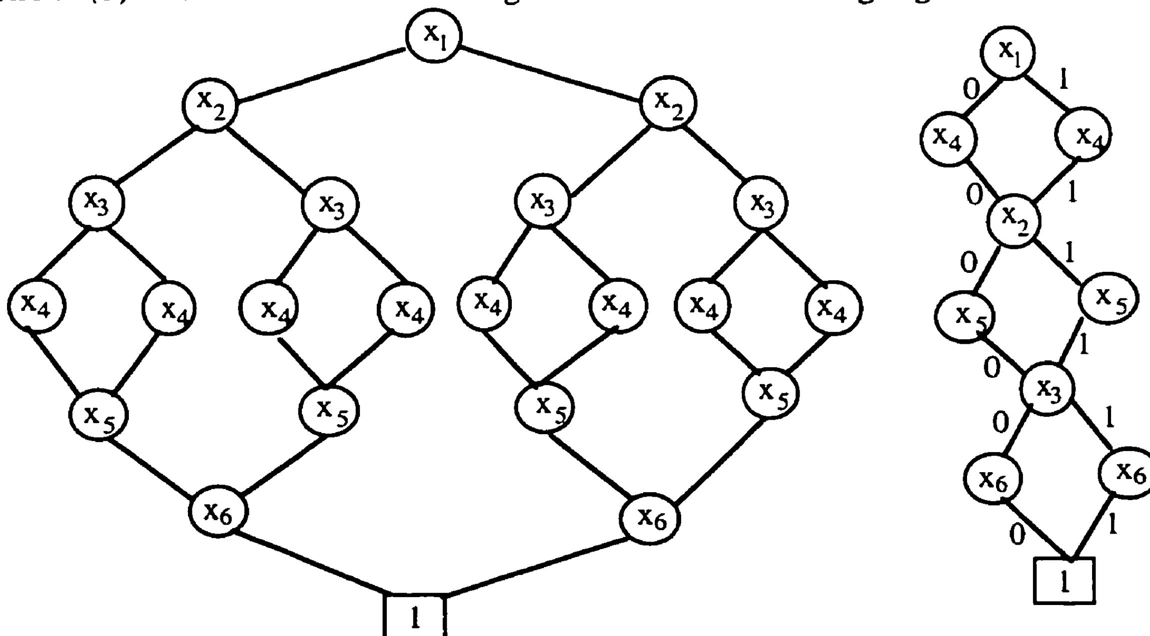


Figure 3.4 Partial-OBDDs of *F(3)* under different orderings

Proposition 3.2 Function *F(k)* has 2^k product terms if *F(k)* is in SOP form. Moreover, any permutation equivalent function of F still has 2^k product terms in SOP form.

Proof It is obvious that *F(k)* has 2^k minterms. However, the distance between any pair of minterms is greater than 1. Therefore no two minterms can form a product term. Therefore if *F(k)* is in SOP form, it has exactly 2^k product terms.

Furthermore, for any two minterms, any permutation does not change the distance between them. Therefore the function *F(k)* has 2^k product terms in SOP form for any permutation.

Proposition 3.3 Function $F(k)$ has $2^{k+1}+2^k-1$ nodes in the OBDD under the standard ordering. However, the optimal OBDD of *F(k)* requires only linear size OBDD.

Proof Under the standard ordering, the partial-OBDD of *F(k)* consists of two parts as shown in the left side of Figure 3.4. Each part is a complete k- variable binary decision tree. Therefore there are $2^{k+1}+2^k-1$ nodes in the OBDD.

Suppose the $2k$ variables are denoted by $x_{k-1}\cdots x_0 y_0 \cdots y_{k-1}$, then the minterms in the function $F(k)$ has the property that $y_i = x_i$. Therefore we have the proposition that $f_{x_i y_i} = f_{\bar{x}_i \bar{y}_i} = f(k-1)$, $f_{x_i \bar{y}_i} = f_{\bar{x}_i y_i} = 0$. Therefore under the ordering $x_{k-1} < y_{k-1} \cdots x_0 < y_0$, the partial-OBDD of *F(k)* is of the form shown in the right side of Figure 3.4. It is of linear size in the number of variables.

OBDD minimization has been studied by many people. Many heuristics have been developed to find an ordering under which functions can have smaller size OBDDs ([17], [2], [18], [20], [21], [22]). Some absolute OBDD minimization algorithms have also been reported [19]. The basic approach of OBDD minimization algorithms is to search permutations of n variables and find the ordering under which a function has the smallest OBDD. The efficiency of the algorithm is determined by the size of the search space. Techniques for reducing the search space are useful. In the next chapter, we introduce a

new Boolean function classification theory. The newly defined single-faced variables will be a useful filter to reduce the search space for OBDD minimization.

Definition 3.14 The sensitivity of a function f is the quotient of the size of a reduced OBDD with respect to a worst variable ordering and the size of a reduced OBDD with respect to a best variable ordering [11].

Theorem 2 [11] The fraction of Boolean functions whose sensitivity is larger than $1+O(n^2 2^{-n/3})$ is bounded by $O(2^{-n/3+\delta n})$ for δ >0.

Based on this theorem, the effect of variable ordering is not important for almost all Boolean functions. In this thesis, we prove that for symmetric functions, their sensitivity is 1, i.e., with respect to any variable ordering, the OBDDs have the same size.

Next we show the relationship between OBDD minimization and function permutation, which was defined in Chapter 2.

Definition 3.15 Two OBDDs G and G' have an *identical graph* if there exists a one-to-one function σ from the vertices of G onto the vertices of G' such that for any vertex v if $\sigma(v)=v'$, then either both v and v' are terminal vertices, or both v and v' are non-terminal vertices with $\sigma(lo(v))=lo(v')$, and $\sigma(hi(v))=hi(v')$.

The identical OBDD graph property is different from OBDD isomorphism. For OBDDs with identical graph, we only require there is a correspondence between the two graphs, no requirement about the labeling of the corresponding vertices. For OBDD isomorphism, we require that the corresponding vertices be labeled by the same variables.

Example 3.6 The two OBDDs shown in Figure 3.5 have identical graphs, but are not isomorphic.

Definition 3.16 A permutation of n variables $M_{n,n} = \begin{pmatrix} x_1 \cdots x_n \\ x_{i_1} \cdots x_{i_n} \end{pmatrix}$ induces a map on OBDDs as follows: for an $OBDD_n(f)$, $M_{n,n}(OBDD_n(f))$ is an OBDD which has the identical graph as $OBDD_n(f)$ with a permutation over the labeling of vertices. For a vertex v with $index(v)$ in $OBDD_n(f)$, the new OBDD $M_{n,n}(OBDD_n(f))$ has the index as

$M_{n,n}(index(v))$.

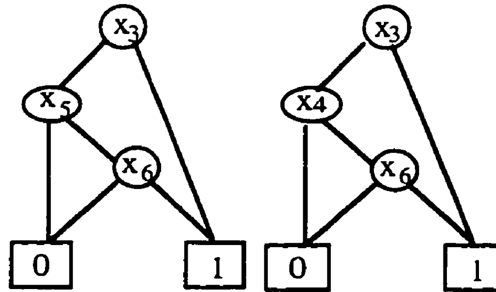


Figure 3.5 OBDDs with identical graphs which are not isomorphic

Proposition 3.4 $M_{n,n}(OBDD_n(f)) = OBDD_{n,M_{n,n}}(M_{n,n}(f))$, therefore

$$\left|OBDD_{n,M_{n,n}}(f)\right| = \left|OBDD_n(M_{n,n}^{inv}(f))\right|.$$

Symmetric functions have identical graphs with respect to any variable ordering.

Proof The function represented by $M_{n,n}(OBDD_n(f))$ is the function $M_{n,n}(f)$. This is true because for every path (and therefore the associated product term), the permutation has been performed. The function represented by an OBDD is the sum of all the product terms associated with all paths ending with 1.

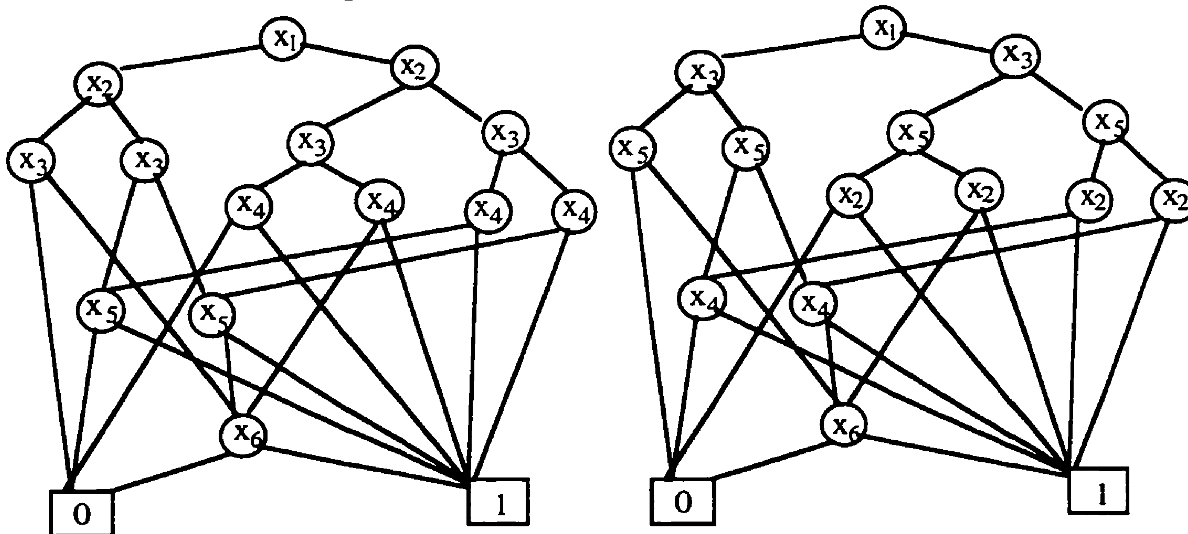


Figure 3.6 Permutation of OBDDs

Example 3.7 The OBDD of $f = x_1x_4 + x_2x_5 + x_3x_6$ under the standard ordering is denoted by $OBDD_6(x_1x_4 + x_2x_5 + x_3x_6)$. Suppose the permutation $M_{6,6}$ is of the form

$$M_{6,6} = \begin{pmatrix} x_1x_2x_3x_4x_5x_6 \\ x_1x_3x_5x_2x_4x_6 \end{pmatrix}, \text{ then}$$

$$M_{6,6}(OBDD_6(x_1x_4 + x_2x_5 + x_3x_6)) = OBDD_{6,\binom{x_1x_2x_3x_4x_5x_6}{x_1x_3x_5x_2x_4x_6}}(M_{6,6}(x_1x_4 + x_2x_5 + x_3x_6))$$
$$= OBDD_{6,\binom{x_1x_2x_3x_4x_5x_6}{x_1x_3x_5x_2x_4x_6}}(x_1x_2 + x_3x_4 + x_5x_6).$$

Both $OBDD_6(x_1x_4 + x_2x_5 + x_3x_6)$ and $OBDD_{6,\binom{x_1x_2x_3x_4x_5x_6}{x_1x_3x_5x_2x_4x_6}}(x_1x_2 + x_3x_4 + x_5x_6)$ are shown in Figure 3.6.

Proposition 3.4 shows that the variable ordering can be translated into the permutation of the functions. Therefore we can always assume the variable ordering for the OBDDs are the standard variable ordering.

Proposition 3.5 For a function $g = M_{n,n}(f)$, where $M_{n,n} \in E_{n,n}$, $|OBDD_n(g)| = \min_{h \in E_{n,n}(f)} \{|OBDD_n(h)|\}$ if and only if $|OBDD_{n,M_{n,n}^{inv}}(f)| = \min_{P \in E_{n,n}} \{|OBDD_{n,P}(f)|\}$.

Proof $|OBDD_n(g)| = |OBDD_n(M_{n,n}(f))| = |OBDD_{n,M_{n,n}^{inv}}(f)|$. On the other hand,

$|OBDD_n(g)| = \min_{h \in E_{n,n}(f)} \{|OBDD_n(h)|\} = \min_{P \in E_{n,n}} \{|OBDD_n(P(f))|\} = \min_{P \in E_{n,n}} \{|OBDD_{n,P^{inv}}(f)|\}$

$= \min_{P \in E_{n,n}} \{|OBDD_{n,P}(f)|\}$. Therefore $|OBDD_{n,M_{n,n}^{inv}}(f)| = \min_{P \in E_{n,n}} \{|OBDD_{n,P}(f)|\}$.

Definition 3.17 A function $g \in E_{n,n}(f)$ is a *minimal function* of $E_{n,n}(f)$ if $|OBDD_n(g)| = \min_{h \in E_{n,n}(f)} \{|OBDD_n(h)|\}$. This is denoted by $g = \min E_{n,n}(f)$.

Based on Proposition 3.5, one class of functions $E_{n,n}(f)$ share the same minimal OBDD function, therefore they also share the same minimal OBDD ordering. The minimization of OBDDs are defined over the set $F(n)/E_{n,n}$. The efficiency of OBDDs comes from the choice of $E_{n,n}(f)$. Accordingly, one would expect two level logic minimization may have better results if we allow minimization over the set $F(n)/E_{n,n}$. This new insight points to a new direction for traditional logic minimization problems, which should be addressed in future research.

Part II

Boolean Function Classification, Minimization, and Application

In this part, we define a Boolean function classification theory and study minimization algorithms. Structures and complexity of OBDDs of single-faced functions and symmetric functions are also studied.

Chapter 4[1] Single-Faced Boolean Functions and the Structure of Their OBDDs

The Boolean function minimization problem is well defined in Part II. In this chapter, we introduce a useful search space filter for Boolean function minimization, which is based on a new Boolean function classification theory. The theory is well defined over the set $F(n)/E_{n,n}$. However, for clarity of presentation, we only state the result over the set $F(n)$. The structure of the OBDDs of single-faced functions is studied. A complexity result on the size of the OBDDs of single-faced functions is presented.

§4.1 A Boolean Function Classification Theory

Definition 4.1 The *dimension of* $f \in F(n)$, denoted by dim(f), is defined as dim(f)=$|\sup(f)|$, i.e., the number of variables in the supporting variable set. Moreover, we define the following sets, where $m<n$.

$C(n) = \{f \in F(n) \mid \dim(f) = n\}$

$L(n) = \{f \in F(n) \mid \dim(f) < n\}$

$F_n(m) = \{f \in F(n) \mid \dim(f) = m\}$

$C(n)$ ($L(n)$) is written as $C[x_1,\cdots,x_n]$ ($L[x_1,\cdots,x_n]$) when variables are considered. It is easy to see that for $n>0$, $L(n) = \bigcup_{m=0}^{n-1} F_n(m)$, $L(n) \neq \varnothing$, and $F(n)=C(n) \cup (\bigcup_{m=0}^{n-1} F_n(m))$.

Definition 4.2 Given a function $f: B^n \to B$ with $\sup(f) = \{x_{i_1}, x_{i_2}, \cdots, x_{i_m}\}$, where $i_1 < i_2 < \cdots < i_m$, $n>m$, the *function restriction map* $R_{m,n}: F_n(m) \to C(m)$ $(n>m\geq 0)$ is defined as $R_{m,n}(f)(y_1,\cdots,y_m) = f_{x_1 \cdots x_{i_1-1} x_{i_1+1} \cdots x_{i_2-1} x_{i_2+1} \cdots x_{i_m-1} x_{i_m+1} \cdots x_n}(x_{i_1}, x_{i_2}, \cdots, x_{i_m})$.

Proposition 4.1 $R_{m,n}(F_n(m)) \subset C(m)$

---

[1] This chapter is based on the paper "Boolean Function Minimization and Classification", which has been submitted to J. of ACM.

The function restriction map $R_{m,n}$ maps functions of n variables to functions with fewer variables. On the other hand, there are well-defined maps from the function set $C(m)$ to the function set $F_n(m)$ $(n>m\geq 0)$.

Definition 4.3 For two variable sets $L_n=\{x_1,\cdots,x_n\}$ and $L_{n+1}=\{x_1,\cdots,x_n,x_{n+1}\}$, a 1-1 map $M_{n+1,n}: L_n \rightarrow L_{n+1}$ is a *basic extension of n variables.* The set of all basic extensions of n variables is denoted by $E_{n+1,n}$. Among them, the map $N_{n+1,n}: L_n \rightarrow L_{n+1}$ is called the *natural extension* of n variables if for $1\leq i\leq n$, $N_{n+1,n}(x_i)=x_i$. We can choose other basic extension as the natural extension as well; such choices do not affect the results presented in this thesis.

For $n>k\geq m>0$, a sequence of variable sets $\{x_1,\cdots,x_k\}$, and a sequence of basic extensions, $M_{k+1,k}$, the composition of n-m basic extensions $M_{k+1,k}$ is a *n-m variable extension*, which is denoted by $M_{n,m}$, i.e., $M_{n,m}=M_{n,n-1}\bullet M_{n-1,n-2}\bullet\cdots\bullet M_{m+1,m}$. A variable extension $M_{n,m}$ can be written as $\begin{pmatrix} x_1x_2\cdots x_m \\ x_{i_1}x_{i_2}\cdots x_{i_m}\end{pmatrix}$, where for $1\leq j\leq m$, $M_{n,m}(x_j)=x_{i_j}$. The set of all such $M_{n,m}$ form the set $E_{n,m}$. Among them, $N_{n,m}=N_{n,n-1}\bullet N_{n-1,n-2}\bullet\cdots\bullet N_{m+1,m}$ is the *n-m natural extension of m variables.* In other words, the *n-m natural extension* of m variables is of the form $N_{n,m}:\{x_1,\cdots,x_m\}\rightarrow\{x_1,\cdots,x_n\}$ such that $N_{n,m}(x_i)=x_i$.

Proposition 4.2 An n-m variable extension $M_{n,m}$: $L_m \rightarrow L_n$ is a one-to-one map. Moreover $M_{n,m}=M_{n,n}N_{n,m}$, where $M_{n,n}\in E_{n,n}$, i.e., every variable extension is the composition of the natural extension and some permutation.

The above proposition shows the important role the natural extension plays. The natural extension of n variables $N_{n+1,n}: L_n \rightarrow L_{n+1}$ induces a Boolean space map from B^n to B^{n+1} as $N_{n+1,n}(\dot{x}_1\dot{x}_2\cdots\dot{x}_n)=\{(\dot{x}_1\dot{x}_2\cdots\dot{x}_n x_{n+1}),(\dot{x}_1\dot{x}_2\cdots\dot{x}_n\bar{x}_{n+1})\}$. The induced map is called the *natural extension of* B^n. A variable extension $M_{n,m}$ also induces a Boolean space map $M_{n,m}: B^m \rightarrow B^n$. The result of such a map can be obtained by the composition rule of maps. The Boolean space map induced by a variable extension $M_{n,m}$ is called a Boolean

space *extension of* B^m *into* B^n, or simply *an extension*. Notice here the extension is still denoted by $M_{n,m}$.

A Boolean space extension $M_{n,m}$ is an onto map. For any $y \in B^m$, $M_{n,m}(y)$ is a n-m dimensional Boolean space, where $n \geq m \geq 0$. The inverse map of a Boolean space extension $M_{n,m}$ is called a *projection map*. When $m=1$, the projection map is a Boolean function.

Definition 4.4 For a function $f \in F[x_1, \cdots, x_n]$, the *natural direct extension* of function f, induced by the natural extension $N_{n+1,n}$, denoted by $N^d_{n+1,n}(f)$, is defined as $(N^d_{n+1,n}(f))(x_1, \cdots, x_n, x_{n+1}) = f(N^{inv}_{n+1,n}(x_1, \cdots, x_n, x_{n+1})) = f(x_1, \cdots, x_n)$.

Based on the composition rule of maps induced by Boolean space extensions, for $M_{n+1,n} = M_{n+1,n+1} N_{n+1,n} \in E_{n+1,n}$,

$$M^d_{n+1,n}(f)(x_1, \cdots, x_n, x_{n+1}) = (M_{n+1,n+1} N_{n+1,n})^d (f)(x_1, \cdots, x_n, x_{n+1})$$
$$= M_{n+1,n+1}(N^d_{n+1,n}(f))(x_1, \cdots, x_n, x_{n+1}) = N^d_{n+1,n}(f)(M_{n+1,n+1}(x_1, \cdots, x_n, x_{n+1}))$$

Similarly, a Boolean space extension $M_{n,m} : B^m \to B^n$ $(n \geq m \geq 0)$ induces a map $M^d_{n,m} : F(m) \to F(n)$. $M^d_{n,m}$ is called a *direct extension of functions*. The set of all such direct extensions is denoted by $E^d_{n,m}$.

Proposition 4.3 For $n > m \geq 0$, if $f \in C(m)$, then $M^d_{n,m}(f) \in F_n(m)$; moreover, $R_{m,n}(M^d_{n,m}(f)) \in E_{m,m}(f)$.

Proof For a direct extension $M^d_{n,m}$ induced by $M_{n,m} = \begin{pmatrix} x_1 x_2 \cdots x_m \\ x_{i_1} x_{i_2} \cdots x_{i_m} \end{pmatrix}$, let $\{x_1, \cdots, x_n\} - \{x_{i_1}, x_{i_2}, \cdots, x_{i_m}\} = \{x_1, \cdots, x_{i_1 - 1}, x_{i_1 + 1}, \cdots, x_{i_2 - 1}, x_{i_2 + 1}, \cdots, x_{i_m - 1}, x_{i_m + 1}, \cdots, x_n\} = \{y_1, \cdots, y_{n-m}\}$, for any function $f \in F(m)$ and a cube $\dot{y}_1 \cdots \dot{y}_{n-m}$, $M^d_{n,m}(f)_{\dot{y}_1 \cdots \dot{y}_{n-m}} = M^d_{n,m}(f)_{y_1 \cdots y_{n-m}} = f$.

Proposition 4.4 For a function $f \in F_n(m)$ $(n > m \geq 0)$, there exists an extension $M^d_{n,m}$ such that $M^d_{n,m}(R_{m,n}(f)) = f$. Therefore $E^d_{n,m}$ $(C(m)) = F_n(m)$.

Corollary 4.1 (1) For $n>0$, $F(n) = C(n) \cup (\bigcup_{m=0}^{n-1} F_n(m)) = C(n) \cup (\bigcup_{m=0}^{n-1} E^d_{n,m}(C(m)))$.

(2) $F_n(m)/E_{n,n} \xrightarrow{R_{m,n}} C(m)/E_{m,m} \xrightarrow{E^d_{n,m}} F_n(m)/E_{n,n}$ is one-to-one and onto.

This proposition provides a well-defined classification of Boolean functions in terms of function dimensions and function equivalent classes.

Example 4.1 The following is the classification of 2-variable functions.

$F(2)=\{x_1\oplus x_2,\overline{x_1\oplus x_2},x_1x_2,x_1\bar{x}_2,\bar{x}_1x_2,\bar{x}_1\bar{x}_2,x_1+x_2,x_1+\bar{x}_2,\bar{x}_1+x_2,\bar{x}_1+\bar{x}_2,x_1,x_2,\bar{x}_2,\bar{x}_1,0,1\}$
$C(2)=\{x_1\oplus x_2,\overline{x_1\oplus x_2},x_1x_2,x_1\bar{x}_2,\bar{x}_1x_2,x_1+x_2,\bar{x}_1\bar{x}_2,x_1+\bar{x}_2,\bar{x}_1+x_2,\bar{x}_1+\bar{x}_2\}$
$L(2)=\{x_1,x_2,\bar{x}_2,\bar{x}_1,0,1\}$
$L(2)=F_2(1)\cup F_2(0)$
$F_2(1)=\{x_1,x_2,\bar{x}_2,\bar{x}_1\}$
$F_2(0)=\{0,1\}$

§4.2 A Refined Classification Theory

In the above we developed a function classification based on the dimension of functions. We refine the classification of the set *C(n)* in this section.

A function value is determined by its supporting variables. However supporting variables have differing roles such as the distinction of "control" inputs and "data" inputs [3]. Some supporting variables alone can determine the function value. For example, if $f(x_1,\cdots,x_n)=x_1g(x_2,\cdots,x_n)$, and the variable x_1 has value 0, then f has value 0 no matter what the other variable values are. However, if $g\in C(n-1)$, then all variables $x_2,\cdots,x_n$ are supporting variables. In the following, we study differences between supporting variables.

Definition 4.5 Given a function $f\in C[x_1,\cdots,x_n]$, a variable x_i is a *0-single-faced variable* of f if $f_{\dot{x}_i}$=0. In this case, $\dot{x}_i$ is a *0-single-faced literal*. Similarly, a variable x_i is a *1-single-faced variable of* f if $f_{\dot{x}_i}$ =*1*. In this case, $\dot{x}_i$ is a *1-single-faced literal*.

Both 0-single-faced variables and 1-single-faced variables are *single-faced variables*. A function is *single-faced* if it has a single-faced variable. All n dimensional single-faced functions form the set *S(n)*. A function $f\in C(n)$ is *double-faced* if $f\notin S(n)$. The set of all double-faced functions is denoted by $D[x_1,\cdots,x_n]$, or *D(n)* if variables are not considered. The distinction of *S(n)* and *D(n)* provides a refined classification of *C(n)*. Functions in *S(n)* have behavior similar to functions in the set *L(n)*, which will be shown below.

It is easy to see that if $f\in C(n)$ has a 0-single-faced literal $\bar{\dot{x}}_i$, then $f=\dot{x}_if_{\dot{x}_i}$ and $f_{\dot{x}_i}\in C(n-1)$. Similarly, if $f\in C(n)$ has a 1-single-faced literal $\bar{\dot{x}}_i$, then $f=\bar{\dot{x}}_i+\dot{x}_if_{\dot{x}_i}$ and

$f_{\bar{x}_i} \in C(n-1)$. The term "single-faced" refers to the fact that the function is constant on one face of the n dimensional hypercube B^n.

Example 4.2 The following is the classification of 1-dimensional functions.
$C(1) = \{x, \bar{x}\}$
$D(1) = \varnothing$
$S(1) = C(1) = \{x, \bar{x}\}$

Example 4.3 The following is the classification of 2-dimensional functions.
$C(2) = \{x_1 \oplus x_2, \overline{x_1 \oplus x_2}, x_1x_2, x_1\bar{x}_2, \bar{x}_1x_2, \bar{x}_1\bar{x}_2, x_1 + x_2, x_1 + \bar{x}_2, \bar{x}_1 + x_2, \bar{x}_1 + \bar{x}_2\}$
$D(2) = \{x_1 \oplus x_2, \overline{x_1 \oplus x_2}\}$
$S(2) = \{x_1x_2, x_1\bar{x}_2, \bar{x}_1x_2, \bar{x}_1\bar{x}_2, x_1 + x_2, x_1 + \bar{x}_2, \bar{x}_1 + x_2, \bar{x}_1 + \bar{x}_2\}$

Proposition 4.5 If a function $f \in C(n)$ has a 0-single-faced literal $\dot{x}_i$, i.e., $f = \dot{x}_i f_{\dot{x}_i}$, then the 0-single-faced variables of $f_{\dot{x}_i}$ are also 0-single-faced variables of *f*. Similarly, if a function $f \in C(n)$ has a 1-single-faced literal $\dot{x}_i$, i.e., $f = \dot{x}_i + \bar{\dot{x}}_i f_{\bar{\dot{x}}_i}$, then all the 1 single-faced variables of $f_{\bar{\dot{x}}_i}$ are also 1-single-faced variables of *f*.

Proof The case for 0-single faced case is easy. We prove the case for 1-single-faced. Let $f = x_1 + \bar{x}_1 f_{\bar{x}_1}(x_2, \cdots, x_n)$. If $f_{\bar{x}_1}(x_2, \cdots, x_n) = x_2 + \bar{x}_2 f_{\bar{x}_1\bar{x}_2}(x_3, \cdots, x_n)$, then

$$f = x_1 + \bar{x}_1x_2 + \bar{x}_1\bar{x}_2 f_{\bar{x}_1\bar{x}_2}(x_3, \cdots, x_n) = x_2 + x_1 + \bar{x}_1\bar{x}_2 f_{\bar{x}_1\bar{x}_2}(x_3, \cdots, x_n).$$

Proposition 4.6 If a function $f \in C(n)$ has a 0-single-faced variable, then all other single-faced variables of *f* are 0-single-faced. Similarly, if a function $f \in C(n)$ has a 1-single-faced variable, then all other single-faced variables of *f* are 1-single-faced.

Proof If x_1 is a 0-single-faced variable of *f*, without losing generality, we assume $f = x_1 f_{x_1}$. In this case, $f_{\bar{x}_1} = 0$. If at the same time, *f* has a 1-single-faced variable x_2, we can assume *f* is of the form $\bar{x}_2 + x_2 f_{x_2}$, then $f_{\bar{x}_1\bar{x}_2} = 1$ implies $f_{\bar{x}_1} \neq 0$. Therefore, *f* could not have any 1-single-faced variables. The same argument can prove the second part of the proposition.

Corollary 4.2 Suppose a function $f \in C(n)$ has k 0-single-faced variables $\{\dot{x}_1, \cdots, \dot{x}_k\}$, then $f = \dot{x}_1 \cdots \dot{x}_k g(x_{k+1}, \cdots, x_n)$. If $f \in C(n)$ has k 1-single-faced variables

$\{\dot{x}_1,\cdots,\dot{x}_k\}$, $f = \dot{x}_1 + \cdots + \dot{x}_k + \bar{\dot{x}}_1 \cdots \bar{\dot{x}}_k g(x_{k+1},\cdots,x_n)$, where $g(x_{k+1},\cdots,x_n) \in C(n-k)$, and $n \geq k > 0$.

Definition 4.6 The map $SR_n : C(n) \to D(m)$ is defined as follows, where $n\text{-}k \geq m$. $SR_n(f) = f$ if $f \in D(n)$. Otherwise,

$$SR_n(\dot{x}_1 \cdots \dot{x}_k g(x_{k+1},\cdots,x_n)) = SR_{n-k}(g(x_{k+1},\cdots,x_n));$$

$$SR_n(\dot{x}_1 + \cdots + \dot{x}_k + \bar{\dot{x}}_1 \cdots \bar{\dot{x}}_k g(x_{k+1},\cdots,x_n)) = SR_{n-k}(g(x_{k+1},\cdots,x_n)).$$

For $f \in C(n)$, $SR_n(f)$ is a double-faced function and it is called the *double-faced core function* of *f*. We let the set $S_n(i)$ denote the set of single-faced functions which have an i dimensional double-faced core function. It is easy to see that $S(n) = \bigcup_{i=0}^{n-1} S_n(i)$; $S_n(1) = \varnothing$ since $D(1) = \varnothing$. In order to make the notation consistent, we assume $D(0)$ is well defined. $D(0) = F(0) = \{1, 0\}$, $S(0) = \varnothing$. Functions in $S_n(0)$ are *complete single-faced functions.*

Example 4.4 For 2 dimensional single-faced functions,
$S(2) = S_2(0) = \{x_1x_2, x_1\bar{x}_2, \bar{x}_1x_2, \bar{x}_1\bar{x}_2, x_1 + x_2, x_1 + \bar{x}_2, \bar{x}_1 + x_2, \bar{x}_1 + \bar{x}_2\}$
$S_2(1) = \varnothing$

Proposition 4.7 For n variables, there are 2^{2n-1} complete single faced functions.

Proof Suppose for m variables, where $m > 1$, there are $N(m)$ complete single-faced functions, then $N(m+1) = 4N(m)$. For a complete single-faced function, each variable can be a 0-single-faced variable or 1-single-faced variable. For both 0-single-faced and 1-single-faced variables, there are also two choices to make the positive literal or negative literal as the single-faced literal. Therefore there are 4 choices for each variable. Moreover, we have $N(1) = 2$. Therefore we have the conclusion.

The number of complete single-faced functions is higher than the number of symmetric functions, which is equal to 2^n. What makes the single-faced functions even more important is the following result.

Definition 4.7 A function *f* is a *complete double-faced* function if for any product term P of dimension greater than 1, f_P is a double-faced function.

Proposition 4.8 If a function f is a complete double faced function, then it has to be the odd or even parity function.

Proof Let $OP(n)$ denote the n-bit odd parity function. Given the function f, we prove the proposition by induction on the number of variables.

For $n=2$, the set $D(2)=\{x_1 \oplus x_2,\ \overline{x_1 \oplus x_2}\}$ consists of only complete double-faced functions. Therefore the conclusion is right.

Assume the conclusion is correct for $k \leq n$. Now we assume $k=n+1$. Since f is a complete double-faced function, therefore for any product term $P=\dot{x}_1 \cdots \dot{x}_{n-1}$, f_P is a double-faced function, i.e., $f_P \in D(2)$. Therefore either $f_P = x_n \oplus x_{n+1}$ or $f_P = x_n \overline{\oplus}\, x_{n+1}$. We define a function $F(x_1,\cdots,x_{n-1})$ of n-1 variables as: for any minterm $P=\dot{x}_1 \cdots \dot{x}_{n-1}$, $F(P)=0$ if $f_P = x_n \overline{\oplus}\, x_{n+1}$; $F(P)=1$ if $f_P = x_n \oplus x_{n+1}$. We have the relation that $f = F(x_1,\cdots,x_{n-1}) \bullet (x_n \oplus x_{n+1}) + \overline{F}(x_1,\cdots,x_{n-1}) \bullet (x_n \overline{\oplus}\, x_{n+1}) = F(x_1,\cdots,x_{n-1}) \oplus (x_n \oplus x_{n+1})$ The new function $F(x_1,\cdots,x_{n-1})$ is also a complete double faced function. If there is a product term Q of dimension greater than 1 such that F_Q is a single-faced function, then the function $f_{Q \cdot x_n \cdot x_{n+1}}$ is a single-faced function, and the dimension of $Q \cdot x_n \cdot x_{n+1}$ is greater than 1. We arrive at a contradiction. By induction, $F = OP(n\text{-}1)$ or $F=\overline{OP(n-1)}$. Therefore either $f = x_n \oplus x_{n+1} \oplus OP(n-1) = OP(n+1)$ or $f = x_n \oplus x_{n+1} \oplus \overline{OP(n-1)}$ $= \overline{OP(n+1)}$.

Corollary 4.3 For a non-complete double-faced function f, there exist product terms P whose dimension is higher than 1 such that the restrictions with respect to those product terms f_P are single-faced functions.

Example 4.5 Consider the function $f = x_1x_2 + x_3x_4 + x_5x_6$. f is not a single-faced function. However, f_{x_i} is a single faced function for all x_i.

The above proposition shows that single-faced functions are commonly encountered. Even if a function is a double-faced function, some of its restrictions may be single-faced. The only complete double-faced functions are the odd and even parity function. For a single-faced function, its double-faced core function again contains some single-faced

restrictions.

The map $SR_n : C(n) \to D(i)$ maps $S_n(i)$ to $D(i)$, resulting in functions with fewer variables and therefore less complexity. Next we show how to obtain the set $S_n(i)$ from $D(i)$.

Definition 4.8 For a function $f \in D[x_1,\cdots,x_n]$, the *natural 0-single-faced extension* of *f*, induced by the natural extension $N_{n+1,n} : \{x_1,\cdots,x_n\} \to \{x_1,\cdots,x_n,x_{n+1}\}$, denoted by $N^0_{n+1,n}(f)$, is defined as $N^0_{n+1,n}(f)(x_1,\cdots,x_n,x_{n+1}) = \dot{x}_{n+1}f(x_1,\cdots,x_n)$. The *natural 1-single-faced extension* of is defined as $N^1_{n+1,n}(f)(x_1,\cdots,x_n,x_{n+1}) = \dot{x}_{n+1} + \bar{\dot{x}}_{n+1}f(x_1,\cdots,x_n)$.

Both the natural 0-single-faced extension and the natural 1-single-faced extension are natural single-faced extensions (denoted by $N^s_{n+1,n}$, i.e., $N^s_{n+1,n}$= $N^0_{n+1,n}$ or $N^1_{n+1,n}$).

The natural 0 (1)-single-faced extension can be combined with permutations to form basic single-faced extensions.

Definition 4.9 A *basic single-faced extension* is defined to be $M_{n+1,n+1}N^s_{n+1,n}(f)$, where $M_{n+1,n+1} \in E_{n+1,n+1}$. It is denoted by $M^s_{n+1,n}$. All the basic single-faced extensions $M^s_{n+1,n} : D(n) \to F(n+1)$ form the set $E^s_{n+1,n}$.

Generally, the composition of a sequence of basic single-faced extensions $M^s_{n,n-i}(f)$= $M^s_{n,n-1}(M^s_{n-1,n-2}(\cdots M^s_{n-i+1,n-i}(f)\cdots))$ is an *n-i* single-faced extension. The set of all such extensions is denoted by $E^s_{n,n-i}$.

Proposition 4.9 For a function $f \in S_n(m)$ $(n>m\geq 0)$, there exists an extension $M^s_{n,m}$ such that $M^s_{n,m}(SR_n(f))$=*f*. Therefore $E^s_{n,m}$ $(D(m))$= $S_n(m)$.

Corollary 4.4 For $n>0$, $S(n) = \bigcup_{i=0}^{n-1} S_n(i) = \bigcup_{i=0}^{n-1} (E^s_{n,i}(D(i))$.

This formula classifies the set $S(n)$ in terms of the dimension of the double-faced core functions. Applying this formula to the classification of Boolean functions, we have a refined Boolean function classification as follows.

Proposition 4.10 For $n>0$, the set of Boolean functions can be classified as follows: $F(n) = \bigcup_{m=0}^{n}\bigcup_{i=0}^{m} E^d_{n,m}[E^s_{m,i}(D(i))]$.

This proposition gives the complete classification of Boolean functions in terms of their double-faced core functions. For any functions, their double-faced core functions are important. This result can be applied in many areas such as logic minimization, testing, symmetry detection and so on.

§4.3 The Structure of Single-faced Functions.

In this section, we study the structure of single-faced functions. We first consider the 0-single-faced function $\dot{x}_i h(x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_n)$, $h \in D(n-1)$. This function has only one single-faced variable x_i, we investigate the edges labeled by $\dot{x}_i$. The term partial-OBDD is simplified as p-OBDD. The variable ordering here is always the standard ordering.

Proposition 4.11 The root graph of variable x_i in the reduced p-OBDD of $\dot{x}_i h(x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_n)$, $h \in D(n-1)$, has the following properties:

(1) all terminal edges are labeled by the single-faced literal $\dot{x}_i$.

(2) every semi-terminal vertex is a unate vertex.

(3) every terminal vertex v has only one semi-terminal vertex *semi(v)*.

Proof Because the function represented by a p-OBDD is the sum of all the product terms associated with all paths in the p-OBDD, all paths in the p-OBDD of $\dot{x}_i h(x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_n)$ ($h \in D(n)$) have to contain an edge labeled by $\dot{x}_i$. Therefore all terminal edges in the root graph of variable x_i are labeled by $\dot{x}_i$. If an edge is not labeled by $\dot{x}_i$, then it is labeled by a variable less than x_i. In the original p-OBDD, this edge must be connected to an edge labeled by a literal greater than x_i or it is not connected to any edge anymore. Therefore we have a path in the p-OBDD which does not contain the single-faced variable x_i.

If a semi-terminal vertex has two children, one of the two edges is labeled by $\bar{x}_i$, contradicting the condition that $\dot{x}_i$ is 0-single-faced literal.

Furthermore, if two semi-terminal vertices v_1 and v_2 have the same child terminal 1 vertex, then the graph rooted in these two vertices are isomorphic. For reduced p-OBDD, this can not happen.

Example 4.6 The following figure shows the root graph of the p-OBDD of $x_i h(x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_n)$.

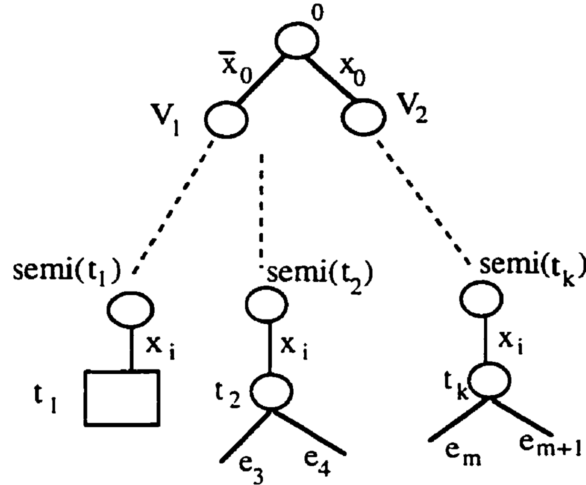


Figure 4.1 The root graph of the p-OBDD of a 0-single-faced function

Root graphs for 1-single-faced functions are more complicated.

Proposition 4.12 If an OPI P of function $f = \dot{x}_i + \bar{\dot{x}}_i[h(x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_n)]$ contains the single-faced literal $\dot{x}_i$, then it does not contain any literals of variables greater than x_i. On the contrary, if P contains a literal of a variable greater than $\dot{x}_i$, then it must contain the literal $\bar{\dot{x}}_i$.

Proof For the function f, $f_{\dot{x}_i} = 1$, therefore $f_{x_1\cdots x_{i-1}\dot{x}_i} = 1$, i.e., $\sup(f_{x_1\cdots x_{i-1}\dot{x}_i}) = \varnothing$. If an OPI P of f contains the single-faced literal $\dot{x}_i$, then it does not contain any literals of variables greater than x_i. Suppose the OPI is of the form $x_1\cdots x_{i-1}x_{i+1}\cdots x_k$, where literals $x_1\cdots x_{i-1}$ are all less than x_i, and literals $x_{i+1}\cdots x_k$ are all greater than x_i, in this case, the function restriction $f_{x_1\cdots x_{i-1}\dot{x}_i}$ has an OPI $x_{i+1}\cdots x_k$, which is not right. Therefore if an OPI contains a literal of a variable greater than $\dot{x}_i$, then it must contain the literal $\bar{\dot{x}}_i$.

Proposition 4.13 All paths in the p-OBDD of $f = \dot{x}_i + \bar{\dot{x}}_i[h(x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_n)]$, $h \in D(n)$, can be divided into three different sets:

(1) the path has all edges labeled by literals of variables less than x_i.

(2) if a path has one edge labeled by $\dot{x}_i$, then all other edges in the path are labeled by literals of variables less than x_i.

(3) if a path has one edge labeled by a variable greater than the single-faced variable x_i, then it must contain an edge labeled by $\bar{x}_i$.

Example 4.7 These three cases are shown in the following example figure, where $x_1 < x_2 < x_3$, and x_3 is the 1-single-faced literal.

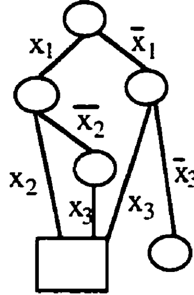


Figure 4.2 Three kinds of paths in the OBDD of a 1-single-faced function

Proposition 4.14 The root graph of variable x_i in the reduced p-OBDD of $\dot{x}_i + \bar{x}_i[h(x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_n)]$, $h \in D(n)$, has the following properties:

(1) all the edges in layers less than $\dot{x}_i$ in the root graph are binary edges.

(2) all edges labeled by $\dot{x}_i$ are terminal edges in the p-OBDD.

(3) all the N-type terminal edges are labeled by $\bar{x}_i$.

(4) every N-type semi-terminal vertex is a binary vertex in the p-OBDD. Moreover, it is connected to one distinct N-type terminal vertex. This edge is labeled by $\bar{x}_i$.

Proof (1) If there is a unate edge in layers less than $\dot{x}_i$, then there is a path of the form $\dot{x}_1\dot{x}_2\cdots\dot{x}_k$ *(k<i)* contained in the p-OBDD of the complement function. Therefore $\dot{x}_1\dot{x}_2\cdots\dot{x}_k x_i \subset \bar{f}$, $\dot{x}_1\dot{x}_2\cdots\dot{x}_k\bar{x}_i \subset \bar{f}$, neither $x_i \subset f$ nor $\bar{x}_i \subset f$ is correct.

(2) by (2) in Proposition 4.13 .

(3) This is a direct corollary of (2).

(4) If it is not a binary edge, then there is a path that contains the literal $\dot{x}_i$ in the complement function. If N-type terminal vertex are not distinct, then the p-OBDD is not reduced.

Example 4.8 The following figure is the root graph of the p-OBDD for $x_i + \bar{x}_i[h(x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_n)]$, where variable $x_k < x_i$, $z < x_i$. The children of t_i is connected to *semi*(t_i).

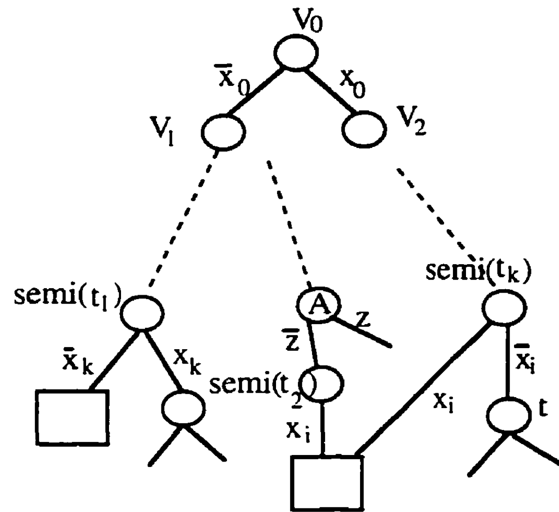


Figure 4.3 The root graph of the p-OBDD of a 1-single-faced function

Based on Proposition 4.11 and Proposition 4.14, algorithms for detecting single-faced variables can be derived.

Moreover, the OBDDs of complete single-faced functions can be determined. The optimal OBDD of a function $f \in S_n(0)$ consists of n non-terminal vertices and two terminal vertices. If a variable is a 0-single-faced variable, then one edge from the vertex labeled by this variable points to 0 terminal vertex, and another edge points to its non-terminal child. If a variable is a 1-single-faced variable, then one edge points to the 1 terminal vertex while another edge points to its non-terminal vertex. The size of the optimal OBDD of a complete single-faced function is always n+2.

Figure 4.4 An example OBDD of a complete single faced function

§4.4 Minimization of Single-faced Functions

Single-faced variables play an important role in Boolean function minimization. We will prove that for SOP representation, factored form representation, and OBDDs, the single-faced function minimization problem can be simplified into the minimization of their double-faced core functions. Based on experience with ESPRESSO [4], the speed of minimization heavily depends on the number of variables. The double-faced core function has fewer variables than the original functions. Therefore the result here can be used to increase the speed of ESPRESSO [4].

We first consider the SOP and factored form minimization problems. Let *MSOP(g)* denote the minimal SOP representation for function *g*, *MF(g)* the minimal factored form representation. We have the following proposition. Here the minimal cost is defined to as minimal number of product terms for SOP, and minimal number of literals for factored forms.

Proposition 4.15 Suppose a function $f \in F(n)$ and $M^d_{n,m} \in E^d_{n,m}$, $M^s_{m,i} \in E^s_{m,i}$ such that $f = M^d_{n,m} M^s_{m,i}(g)$, where $g \in D(i)$; then we have

(1) $MSOP(f) = M^d_{n,m} M^s_{m,i}(MSOP(g))$.

(2) $MF(f) = M^d_{n,m} M^s_{m,i}(MF(g))$

Proof Result (1) can be easily proven by the observation that a 1-single-faced literal is an EPI; while for a 0-single-faced literal, every product term has to contain this literal. Therefore (1) is true.

To prove (2), we prove that for any factored form g of the 1-single-faced function $f = x_1 + \bar{x}_1 f_{\bar{x}_1}(x_2,\cdots,x_n)$, we can find a new factored form $h = x_1 + \bar{x}_1 \bullet Tr(g)$ with less literals and representing the same function *f*. The factored form *Tr(g)* is obtained by the following algorithm.

$$Tr(0) = 0$$
$$Tr(1) = 1$$
$$Tr(\bar{x}_1 \bullet g_1 + g_2) = Tr(g_1) + Tr(g_2)$$
$$Tr(x_1 \bullet g_1 + g_2) = Tr(g_2)$$
$$Tr(g_1 + g_2) = Tr(g_1) + Tr(g_2)$$
$$Tr(g_1 \bullet g_2) = Tr(g_1) \bullet Tr(g_2)$$

The factored form h has fewer or equal number literals than g. Because g contains the literal x_1 and $\bar{x}_1$, while *Tr(g)* does not, *Tr(g)* has at least 2 literals fewer than g. Expanding the factored form g and *Tr(g)* into SOP form, let *SOP(g)* denote the expanded SOP form of the factored form g, then we have $SOP(g) = \bar{x}_1 \bullet (SOP(Tr(g)) + x_1$. Therefore h and g represent the same function *f*. The 0-single-faced case can be proven similarly.

According to the result in Chapter 3, the OBDD minimization problem is equivalent to finding the minimal function such that its OBDD is of the minimal size under the standard ordering. We therefore study the minimal functions for single-faced functions in the set $S_n(i)$. In particular we look at $S_{n+1}(n)$. Results about general cases of $S_n(i)$ can be obtained from the results about $S_{n+1}(n)$ by induction. For clarity, we redefine some notation here. The variable sets are $L_n = \{x_1,\cdots,x_n\}$ and $L_{n+1} = \{x_0, x_1,\cdots,x_n\}$. The notation $N^s_{n+1,n}$ refers to $N^0_{n+1,n}$ or $N^1_{n+1,n}$, which are defined as $\dot{x}_0 f(x_1,\cdots,x_n)$ or $\dot{x}_0 + \bar{x}_0 f(x_1,\cdots,x_n)$ accordingly.

Theorem 3 Let $f \in D(n)$, $\min E_{n+1,n+1}(N^s_{n+1,n}(f)) = N^s_{n+1,n}(\min E_{n,n}(f))$.

Proof The set $E_{n+1,n+1}$ can be divided into two disjoint subsets $E_{n,n}(x_0)$ and D_{n+1}. A permutation $P \in E_{n,n}(x_0)$ if $P(x_0)=x_0$. If $P \in E_{n,n}(x_0)$, $P=\begin{pmatrix} x_0 x_1 \cdots x_n \\ x_0 x_{i_1} \cdots x_{i_n} \end{pmatrix}$ induces a map $P_{n,n}=\begin{pmatrix} x_1 \cdots x_n \\ x_{i_1} \cdots x_{i_n} \end{pmatrix}$, $P_{n,n} \in E_{n,n}$. One can see the 1-1 correspondence between $E_{n,n}(x_0)$ and $E_{n,n}$. Therefore, $\min\{[E_{n,n}(x_0)](N^s_{n+1,n}(f))\} = \min\{E_{n,n}(N^s_{n+1,n}(f))\} = N^s_{n+1,n}(\min E_{n,n}(f))$.

On the other hand, we consider 0-single-faced extensions as an example. For any
$P_{n+1,n+1}=\begin{pmatrix} x_0 x_1 \cdots x_n \\ x_{i_0} x_{i_1} \cdots x_{i_n} \end{pmatrix} \in D_{n+1}$,
$P_{n+1,n+1}(N^s_{n+1,n}(f))(x_0,x_1,\cdots,x_n) = P_{n+1,n+1}(\dot{x}_0 f(x_1,\cdots,x_n)) = \dot{x}_{i_0} f(x_{i_1},\cdots,x_{i_n})$. We define a permutation $\tilde{P}_{n,n}=\begin{pmatrix} x_0,x_1,\cdots,x_{i_0-1},x_{i_0+1},\cdots,x_n \\ x_{i_1},x_{i_2}\cdots,x_{i_{i_0}},x_{i_{i_0+1}},\cdots,x_{i_n} \end{pmatrix}$, then

$P_{n+1,n+1}(N^s_{n+1,n}(f))(x_0,x_1,\cdots,x_n) = \dot{x}_{i_0} f(x_{i_1},\cdots,x_{i_n}) = \dot{x}_{i_0}[\tilde{P}_{n,n}(f)(x_0,x_1,\cdots,x_{i_0-1},x_{i_0+1},\cdots,x_n)]$

Let $Q_{n+1,n+1}=\begin{pmatrix} x_0,x_1,\cdots,x_{i_0-1},x_{i_0},x_{i_0+1},\cdots,x_n \\ x_1,x_2,\cdots,x_{i_0},x_0,x_{i_0+1},\cdots,x_n \end{pmatrix}$, then $Q_{n+1,n+1} \bullet P_{n+1,n+1}(x_0)= x_0$.

Moreover, $Q_{n+1,n+1} \bullet P_{n+1,n+1}(N^s_{n+1,n}(f)) = \dot{x}_0[\tilde{P}_{n,n}(f)(x_1,\cdots,x_{i_0-1},x_{i_0}x_{i_0+1},\cdots,x_n)]$.

Lemma 1 For any function $h \in D(n)$,

$$|OBDD_{n+1}(\dot{x}_i[h(x_0,x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_n)])| \ge$$
$$|OBDD_{n+1}(\dot{x}_0[h(x_1,\cdots,x_{i-1},x_i x_{i+1},\cdots,x_n)])|.$$

Similarly,

$$|OBDD_{n+1}(\dot{x}_i+\bar{\dot{x}}_i[h(x_0,x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_n)])| \ge$$
$$|OBDD_{n+1}(\dot{x}_0+\bar{\dot{x}}_0[h(x_1,x_2,\cdots,x_i,x_{i+1},\cdots,x_n)])|$$

Based on this lemma, let $h=\tilde{P}_{n,n}(f)$, we have $|OBDD_{n+1}Q_{n+1,n+1} \bullet P_{n+1,n+1}(N^s_{n+1,n}(f))|$ $\le |OBDD_{n+1}P_{n+1,n+1}(N^s_{n+1,n}(f))|$, where $Q_{n+1,n+1} \bullet P_{n+1,n+1} \in E_{n,n}(x_0)$. Therefore

$\min(E_{n+1,n+1}(N^s_{n+1,n}(f)) = \min\{[E_{n,n}(x_0) \cup D_{n+1}](N^s_{n+1,n}(f))\}$

$= \min\{[E_{n,n}(x_0)](N^s_{n+1,n}(f))\} = N^s_{n+1,n}(\min\{E_{n,n}(f)\})$.

Now we prove the lemma for the 0-single-faced case and 1-single-faced case.

Proposition 4.16 Suppose the root graph of variable x_i in the reduced p-OBDD of $\dot{x}_i h(x_0,x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_n)$, $h \in D(n)$, has root vertex v_0, the set of terminal vertices is $\{t_1,\cdots,t_k\}$, t_i has *semi*(t_i) as its semi-terminal vertex. The edge from *semi*(t_i) to t_i is labeled by variable $\dot{x}_i$, then the p-OBDD of the function $\dot{x}_0[h(x_1,\cdots,x_{i-1},x_i x_{i+1},\cdots,x_n)]$ can be obtained by the following procedure.

(1) Create a new root vertex v, label the edge connecting v to v_0 by $\dot{x}_0$.

(2) Connect all the child or children of vertex t_i to vertex *semi*(t_i). If t_i is a terminal vertex in the p-OBDD, then change *semi*(t_i) to a terminal vertex.

(3) Increase the labeling of layers 0 to layer *i-1* by 1.

Proof This can be proven by enumerating all the paths.

Example 4.9 The above process can be shown by the following figure, where Figure 4.5 (I) is the p-OBDD for $x_i h(x_0,x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_n)$. Figure 4.5 (II) is the p-OBDD for $x_0[h(x_1,\cdots,x_{i-1},x_i x_{i+1},\cdots,x_n)]$.

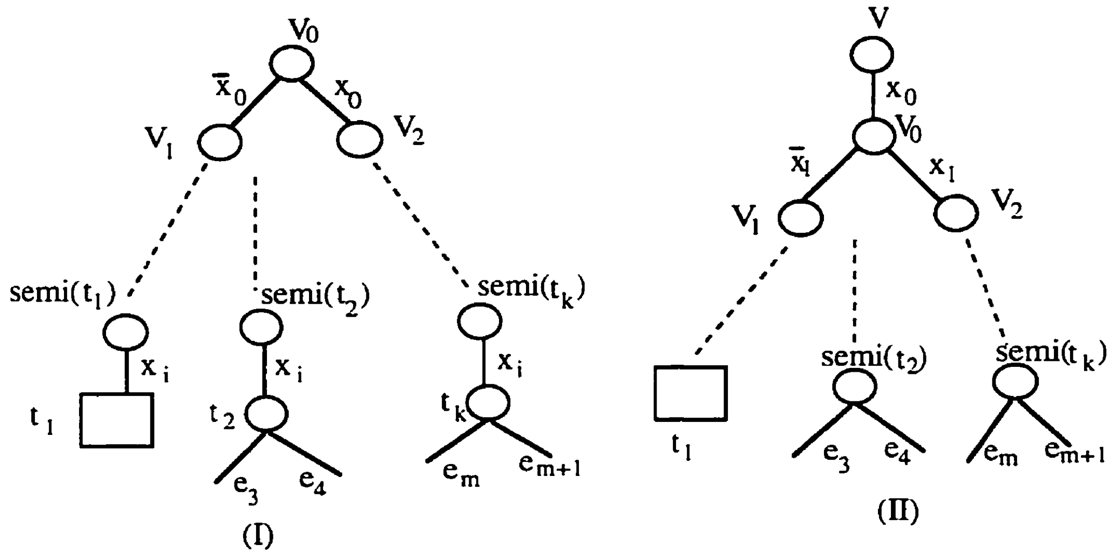


Figure 4.5 The p-OBDDs of different 0-single-faced variables

By counting the number of vertices in the above figures, Lemma 1 for 0-single-faced functions follows easily.

Proposition 4.17 Suppose the root graph of variable x_i in the reduced p-OBDD of $\dot{x}_i + \bar{x}_i[h(x_0,x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_n)]$, $h \in D(n)$ has root vertex v_0. The set of N-type

terminal vertices is $\{t_1, \cdots, t_k\}$. For each t_i, its semi-terminal vertex is $semi(t_i)$. The edge from $semi(t_i)$ to t_i is labeled by literal $\bar{x}_i$, then the following procedure creates the p-OBDD for the function $\dot{x}_0 + \bar{x}_0[h(x_1, x_2, \cdots, x_i, x_{i+1}, \cdots, x_n)]$.

(1) Create a new root vertex v, the edge connecting v to v_0 is labeled by literal $\bar{x}_0$. The edge connecting v to the terminal vertex is labeled by $\dot{x}_0$.

(2) Delete the children of $semi(t_i)$, and connect the children of t_i to $semi(t_i)$.

(3) In the root graph, delete all unate edges and edges that have one unate vertex,

(4) Increase the labeling of layers 0 to layer *i-1* by 1.

Example 4.10 The above process can be shown in the following figure, where Figure 4.6 (I) is the p-OBDD for $x_i + \bar{x}_i[h(x_0, x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n)]$. Figure 4.6 (II) is the p-OBDD for $x_0 + \bar{x}_0[h(x_0, x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n)]$, where variable $x_k < x_i$. In Figure 4.6 (II), the unate edge labeled by x_i is deleted, so the edge labeled by $\bar{z}$ since this edge contains one unate vertex, where z is a variable less than x_i. The children of t_i i s connected to $semi(t_i)$.

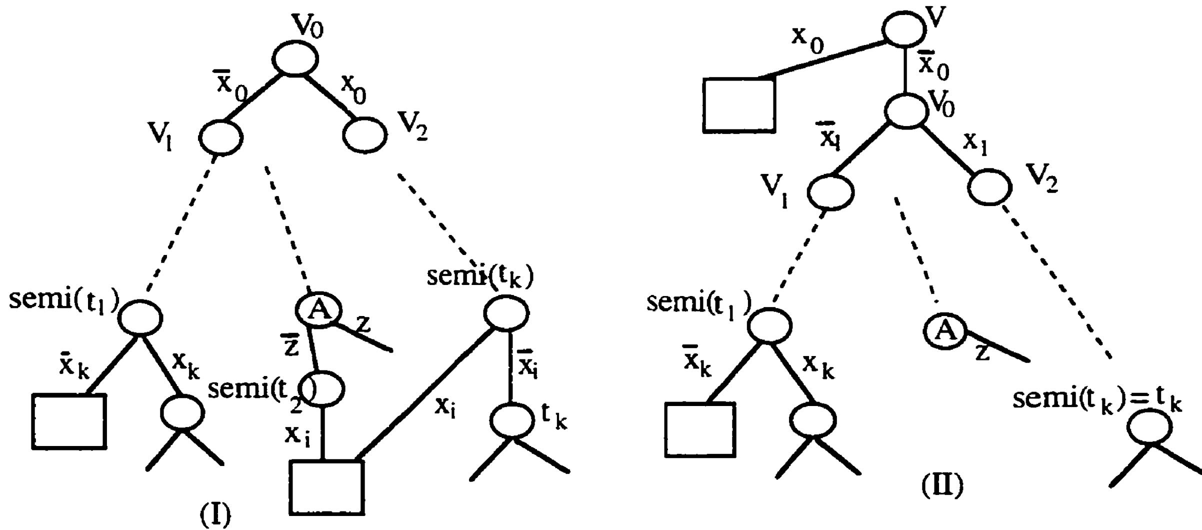


Figure 4.6 p-OBDDs of two different 1-single-faced variables

By counting the vertices in Figure 4.6 (I) and Figure 4.6 (II), Lemma 1 for 1-single-faced functions can be proven. Therefore Theorem 1 is proven.

Based on Theorem 3, better algorithms for OBDD minimization can be developed. For a single-faced function, the best ordering is always to select the single-faced variable as the least variable. This is the theoretical realization of the experience in [3] that "it is generally wise to pick a variable ordering that places the control input variables before the data input variables". Single-faced variables can be viewed as control variables.

In this chapter, we have shown that single-faced variables can be used as a filter for SOP minimization, factored form minimization, and OBDD minimization. A Boolean function classification theory based on single-faced variables is also developed.

Chapter 5[2] OBDD Structure and Complexity of Symmetry Boolean Functions

In this chapter, as an application of the single-faced function classification, we study the OBDD structure of symmetric functions. A new algorithm for symmetry detection is also presented. The structure of symmetric OBDD can also be used to obtain the size of the OBDD of symmetric functions.

§5.1 Introduction

Symmetry is a useful property in logic synthesis, logic optimization, and technology mapping. For symmetric functions, there are special logic synthesis procedures that can improve the results of the design [24], [25], [30]. Symmetry also improves the efficiency of technology mapping and equivalence testing [23], [26], [27].

In order to effectively use symmetry, one has to detect symmetry fast. A naive method for symmetry checking is to test equality of the function restrictions $f_{x_i \bar{x}_j} = f_{\bar{x}_i x_j}$ for all variables x_i and x_j. In this algorithm, all the function restrictions have to be generated. Although this method is popular, creating the necessary cofactor functions is very time consuming [28].

Symmetry of functions represented by OBDDs can be detected more efficiently by preprocessing based on the structure of the OBDDs [28] [43]. The symmetry detection algorithm in [28] avoids generating OBDDs using preprocessing. Using properties of counting the satisfying set or the structure of the OBDDs, asymmetric pairs of variables are

[2] This chapter is based on the paper "Algorithms for and structure of symmetric OBDD", which has been submitted to IEEE. Trans. on Computers.

detected. However the drawback of this method is clear. It does not avoid use of the naive method, i.e., the generation of new OBDDs to test symmetry.

In this chapter, we study the OBDD structure of symmetric functions. Single-faced functions play an important role in the structure of the symmetric functions. A new algorithm for symmetry detection is also presented. The algorithm eliminates the generation of new OBDDs. Symmetry is detected by the structure of the OBDDs. The complexity of the algorithm is equivalent to a depth-first search of the OBDD. The structure of symmetric OBDD can also be used to obtain the size of the OBDD of symmetric functions.

§5.2 The Structure of Symmetric Functions

Proposition 5.1 If a single-faced function is a symmetric function, then it has to be of the form: $\dot{x}_1 \bullet \cdots \bullet \dot{x}_n$ or $\dot{x}_1 + \cdots + \dot{x}_n$, i.e., a symmetric single-faced function is a complete single-faced function.

Proof First we consider a 0-single-faced function *f*. Suppose *f* is of the form $\dot{x}_1 f(x_2, \cdots, x_n)$. Because the function is symmetric, for any variable x_i, it has to be the form $\dot{x}_i f(x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n)$. Therefore it has to be the form $\dot{x}_1 \bullet \cdots \bullet \dot{x}_n$.

Similarly we can prove that a 1-single-faced function has to be the form $\dot{x}_1 + \cdots + \dot{x}_n$.

Proposition 5.2 A function *f* is a symmetric function of n variables if and only if it is one of the following three cases.

(1) the constant 0 or 1.

(2) One function restriction f_{x_1} or $f_{\overline{x}_1}$ is constant 0 or 1, i.e., *f* can be of the form $x_1 \cdots x_n$, $\overline{x}_1 \cdots \overline{x}_n$, $x_1 + \cdots + x_n$, or $\overline{x}_1 + \cdots + \overline{x}_n$.

(3) Neither of the function restrictions f_{x_1} or $f_{\overline{x}_1}$ is constant, then

(i) $f_{\overline{x}_1 x_2} = f_{x_1 \overline{x}_2}$, and

(ii) both f_{x_1} and $f_{\overline{x}_1}$ are symmetric on *n-1* variables $\{x_2, \cdots, x_n\}$.

Definition 5.1 An OBDD is called an *n-triangle* if it is as shown in Figure 5.1. An *n-triangle* consists of n non-terminal nodes labeled by variables $x_{i+1}\cdots x_{i+n}$. All non-terminal vertices have edges labeled by c (either 0 or 1) pointing to one terminal, and all other edges are connected according to the increasing ordering of the variables. The path $x_{i+1}\cdots x_{i+n}$ is called the *n-boundary* of the *n-triangle*. The single edge from x_{i+1} to the terminal vertex is called the *1-boundary* of the triangle. The vertex x_{i+1} is the *top* of the triangle. A 1-triangle is just one variable.

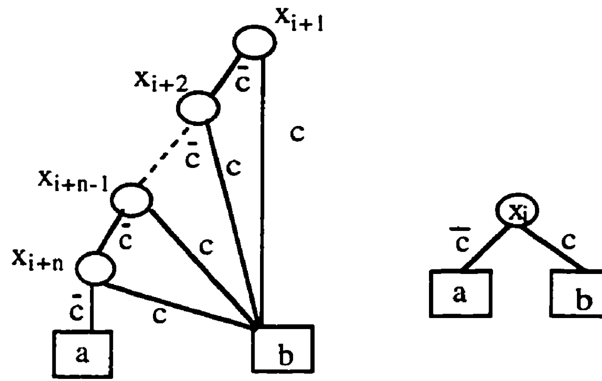


Figure 5.1 n-triangle and 1-triangle

Proposition 5.3 The OBDD of single-faced symmetric functions $x_1\cdots x_n$, $\bar{x}_1\cdots\bar{x}_n$, $x_1+\cdots+x_n$, and $\bar{x}_1+\cdots+\bar{x}_n$, are *n-triangles* with top vertex labeled by x_1.

The rest of this section focuses on the structure of the OBDD of symmetric functions. Single-faced symmetric functions and n- triangles play an important role in determining the structure of the OBDD of symmetric functions.

Definition 5.2 In an OBDD, if a vertex is the root of an n-triangle, then the vertex is called a *single-faced vertex.* Vertices are *double-faced vertices* if they are not single-faced.

Every path in the OBDD of a symmetric function contains at least one single-faced vertex.

Definition 5.3 For an OBDD of a symmetric function f, the first (least) single-faced vertex in the right-most path represents a single-faced symmetric function. This function is called the *right single-faced function* of f.

The right single-faced function of a symmetric function f impacts on the structure of the OBDD of f.

Definition 5.4 An *(m,k)* 0-1 grid is a directed graph consisting of $m*k$ vertices with coordinate (x,y) satisfying the conditions $1 \le x \le m$ and $1 \le y \le k$. Moreover, the vertex (x, y) is connected to $(x+1, y)$ for $1 \le x \le m-1$ and $(x, y+1)$ for $1 \le y \le k-1$. All the edges from (x, y) to $(x+1, y)$ are labeled by 0, and all the edges from (x, y) to $(x, y+1)$ are labeled 1. Moreover, each vertex (x,y) is labeled by a variable x_i such that $i = x + y - 2 + i_0$, where x_{i_0} is the label for the node $(1,1)$.

We define those vertices where $x=1$ or m, or $y=1$ or k as boundary vertices. Among them, (m, y) form the $x=m$ boundary, and the (x, k) form the $y=k$ boundary. Similarly we can define the $x=1$ and $y=1$ *boundary*. The vertex $(m,1)$ is called the *left-corner*, vertex $(1, k)$ is the *right corner*.

Example 5.1 The Figure 5.2 shows the (6,6) 0-1 grid in different positions.

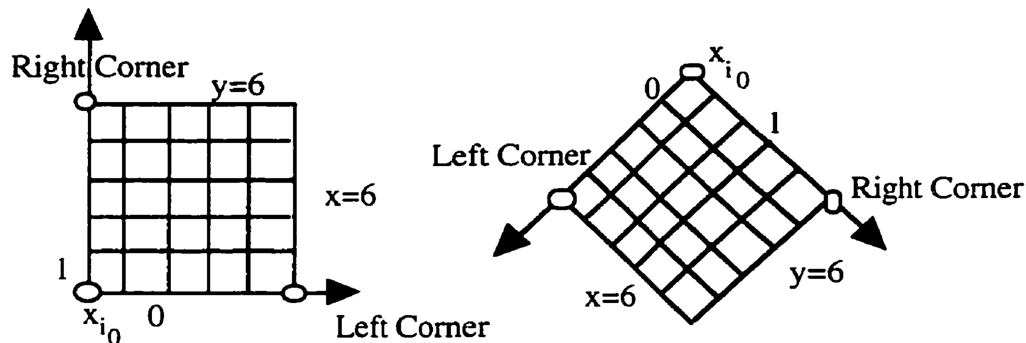


Figure 5.2 (6,6) 0-1 grid.

0-1 grids can be connected to m-triangles to form new graphs. An m-triangle can be connected to a 0-1 grid in two ways.

Definition 5.5 The *first kind of connection* of an *(m,k)* 0-1 grid with an *m-triangle* is defined by a map from the $y=k$ boundary of the grid to the m-boundary of the triangle, such that the vertices have a 1-1 correspondence, and the right corner of the grid is mapped to the top of the triangle.

Definition 5.6 The *second kind of connection* of a 0-1 grid with an *m-triangle* is defined by a map from the $y=k$ boundary of a $(2, k)$ grid to the l-boundary of the triangle, such that the vertices have a 1-1 correspondence, and the right corner of the grid is mapped to the top of the triangle.

Example 5.2 Figure 5.3 shows the first kind of connection of a (6,6) 0-1 grid with a 6-triangle.

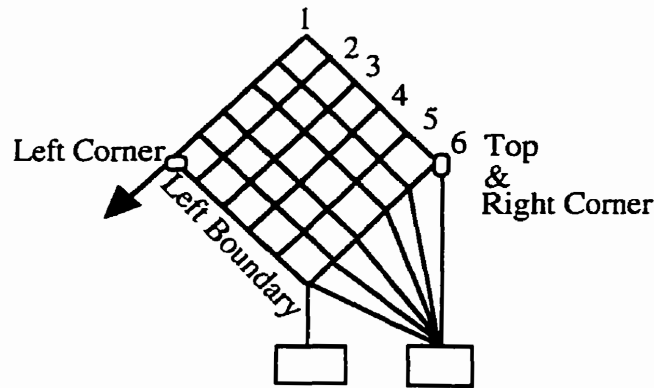


Figure 5.3 First kind of connection.

Example 5.3 The second kind of connection between a (2, 6) 0-1 grid and a 2-triangle is shown in Figure 5.4.

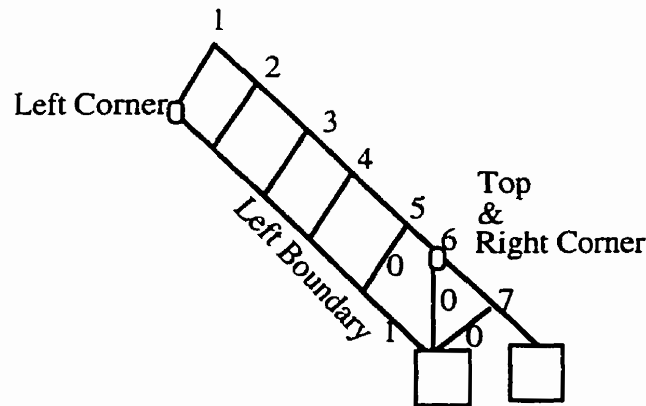


Figure 5.4 Second kind of connection

Definition 5.7 Graphs formed by connecting 0-1 grids together with m-triangles are called *connection graphs.* The *left corner* of the connection graph is the left-corner of the 0-1 grid. The $x=m$ boundary of the 0-1 grid is the *left-boundary* of the connection graph.

Definition 5.8 An OBDD is said to *contain a DAG* if every node and edge in the DAG is contained in the OBDD.

Proposition 5.4 For an n variable symmetric function with an m dimensional right single-faced function, its OBDD contains the connection graph of an *(m, n-m+1) 0-1* grid and an m-triangle, or the connection graph of a *(2, n-m+1)* 0-1 grid and a m-triangle.

This connection graph is called the *connection graph of the OBDD.*

From the connection of the grid and triangle, one can see that the nodes inside the grid all have two children pointed to fixed directions. The only vertices that have only one child are the vertices on the left boundary of the connection graph. The remaining structure of the OBDD is determined by the children of the vertices in the left boundary of the connection graph. One further step can show the children of the vertices in the left boundary are totally determined by one subgraph. We need the following concept.

Definition 5.9 For an OBDD of a symmetric function *f*, the 0-child of the left-corner of the connection graph represents a symmetric function. This function is called the *left tail function* of *f*.

Proposition 5.5 If a symmetric function has an m dimensional right single-faced function, then the left tail function is an *n-m* dimensional symmetric function, or an *n-2* dimensional symmetric function.

Proposition 5.6 Suppose the vertices in the left boundary of the connection graph in the OBDD are labeled by variables $x_k, \cdots, x_{m_1}$, while the right-most path of the OBDD of the left tail function consists of vertices labeled by variables $x_{k+1}, \cdots, x_{m_2}$, then the OBDD of the symmetric function can be obtained by the following means: connect the nodes x_i in the left boundary to the node x_{i+1} in the right-most path, or to a terminal node if x_{i+1} does not exist in the right-most path.

Example 5.4 Figure 5.5 shows the connection.

Figure 5.5 The connection of the left-tail function with the connection graph.

The above process shows that the OBDD of a symmetric function can be constructed by the OBDDs of the left tail function and the right single-faced function. Therefore the OBDD of a symmetric function is totally determined by its right single-faced function and its left tail function.

Example 5.5 The construction of the parity function by the right single faced function and the left tail function.

Figure 5.6 The Odd-even parity function

In Figure 5.6, the node marked by C is isomorphic to the node marked by A, B is isomorphic to D, and E is isomorphic to F. Therefore the reduced OBDD is actually as shown in Figure 5.7.

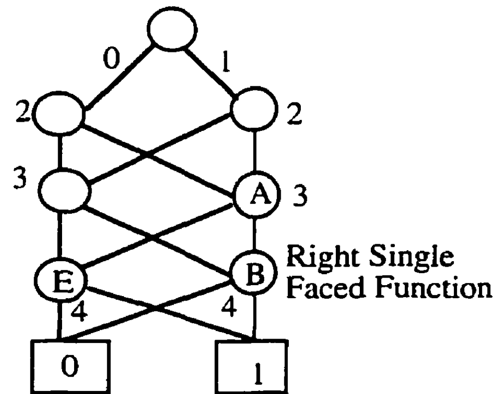


Figure 5.7 The reduced OBDD

The final result in this paper regards the size of OBDDs of symmetric functions. Proposition 5.5 also implies the following result.

Proposition 5.7 There are symmetric functions which need $\Omega(n^2)$ nodes in the OBDD.

Proof Every symmetric function can be represented by a not necessarily reduced binary decision diagram of size $O(n^2)$. This result was mentioned in [1]. On the other hand, the connection graph contained in the OBDD can have at least $m*(n-m+1)$ nodes. When $m=n/2$ or $m=(n+1)/2$, the size $m*(n-m+1) > n^2/4$. Therefore the proposition is correct.

§5.3 Symmetry Detection Algorithm

Proposition 5.2 plus Proposition 5.3 gives the following symmetry detection algorithm, where B is the OBDD we want to test for symmetry, and n is the number of variables. BDD_Then(B) is the right branch of B. BDD_Else(B) is the left branch of B. The algorithm returns true if B represents a symmetric function.

Algorithm SDet-2 (B, n)

$t1$=BDD_Then(B);

$t2$=BDD_Else(B);

If both $t1$ and $t2$ are not constant

```
If(BDD_Else(t1)==BDD_Then(t2) )
        If SDet-2(t1, n-1) && SDet-2(t2, n-1)
                Return(True);
        Else Return(False);
Else
        Return(False).
```

Else Verify *t1* or *t2* is the *n*-boundary of an *n*-triangle.

This algorithm is a one-pass search of the OBDD. Once a non-symmetric branch is found, the result is returned.

We have implemented Algorithm SDet-2 within the SIS package [31]. Tests were done on LGSynth91 benchmarks using a SPARC 10 machine. Results are summarized in Table 1. Time is measured in seconds. The time is that required to test the symmetry of OBDDs after their construction. For those benchmarks not listed in the table, the time is always less than one second.

Table 5.1 CPU Time

Name	**Time**	**Name**	**Time**	**Name**	**Time**
C1355	1.3	dalu	0.1	pair	0.5
C1908	0.5	des	0.4	parity	0
C432	0.7	i8	0.1	rot	0.3
C499	1.3	i10	19.3	too_large	0.1
C5315	0.5	k2	0.1		

Part III New Data Structures

In Part I, we introduced various data structures for Boolean functions and defined the Boolean function minimization problem. In Part II, we showed a useful filter for Boolean function minimization for many data structures. However, we will show in Chapter 6 that in general Boolean function minimization does not make much difference. Almost all Boolean functions have exponential size OBDDs. Therefore we propose some new data structures in Chapter 7. The new data structures can unify many existing data structures. Some functions with exponential size BDDs and SOPs have constant size representation in the new data structure.

Chapter 6 Some Hard Examples and More Data Structures

§6.1 Hard Examples

OBDDs have proven to be a very successful data structures. Many practical Boolean functions possess compact (polynomial size) OBDD-representations. On the other hand, however, there are many important functions without such succinct representations. For example, there exist Boolean functions such as the FHS-function [12], integer multiplication, hidden weighted bit function (HWB) [2], and indirect storage access function [12] that, for no variable ordering, can be represented by OBDD's of polynomial size. In the following, we show some of those examples.

Example 6.1 The function f_n has $2n+\lceil \log(n) \rceil$ inputs, corresponding to variables $a_0,\cdots,a_{n-1},b_0,\cdots,b_{n-1}$ and $mux_1,\cdots,mux_{\lceil \log(n) \rceil}$. $f=g_i$ if $value(mux_1,\cdots,mux_{\lceil \log(n) \rceil})=i$, where function $value(mux_1,\cdots,mux_{\lceil \log(n) \rceil})$ returns the integer value of the input binary combination. The function g_i is defined as $g_i=\sum_{j=0}^{n-1}(a_j b_{(j+i) \bmod n})$ for $0 \leq i < n$. Each function g_i has n product terms. There are n such functions, and each function is ANDed by an input combination over the mux_i's, resulting in the f function having n^2 product terms.

When $n=4$, the function is of the form:

$$f_4=\bar{x}\bar{y}(a_0b_0+a_1b_1+a_2b_2+a_3b_3)+\bar{x}y(a_0b_1+a_1b_2+a_2b_3+a_3b_0)$$
$$+x\bar{y}(a_0b_2+a_1b_3+a_2b_0+a_3b_1)+xy(a_0b_3+a_1b_0+a_2b_1+a_3b_2)$$

Proposition 6.1 ([8]) The OBDD of function f_n has $\Omega(2^{n/2})$ vertices under any possible variable ordering.

The function f_n also serves as the example of functions with polynomial size SOP form and exponential size OBDD representation. In Chapter 3, we showed examples with

exponential size SOP form representation and linear size OBDDs. Therefore SOP form and OBDD are not totally comparable, though statistically considering the application encountered in VLSI design, OBDDs have better performance.

Example 6.2 The *hidden weighted bit* function is an example that requires an OBDD of exponential size, but has a VLSI implementation with low area-time complexity. This function has n inputs: $L_n = \{x_1, \cdots, x_n\}$. For input assignment $x = \dot{x}_1 \cdots \dot{x}_n$, define "weight" to be the number of inputs set to 1. That is $wt(x) = \sum_{j=1}^{n} \dot{x}_j$. The hidden weighted bit function selects the *ith* input, where $HWB(x) = \begin{cases} \dot{x}_i, i = wt(x) > 0 \\ 0, wt(x) = 0 \end{cases}$

Proposition 6.2 ([3]) Any OBDD representation of HWB requires $\Omega(1.14^n)$ vertices.

Example 6.3 (multiplier function) The multiplier function is defined as below. We assume $x_0, \cdots, x_{n-1}$ are binary values. We write $< x_{n-1}, \cdots, x_0 >$ to denote the n-bit unsigned integer $\sum_{i=0}^{n-1} 2^i x_i$. Given input variables $\{x_i\}$ and $\{y_i\}$, define $\{z_i\}$ such that

$$< z_{2n-1}, \cdots, z_0 >=< x_{n-1}, \cdots, x_0 > \times < y_{n-1}, \cdots, y_0 >$$

Then $$< z_{2n-1}, \cdots, z_0 >= \sum_{i=0}^{n-1} 2^i \times < x_i \wedge y_{n-1}, x_i \wedge y_{n-2}, \cdots, x_i \wedge y_0 >.$$

We denote the function z_i of the n-bit multiplier as $M(n,i)(x_{n-1}, \cdots, x_0, y_{n-1}, \cdots, y_0)$.

Proposition 6.3 ([3]) For the function *M(n, n-1)*, any VLSI implementation has area-time complexity $\Omega(n^2)$, and any OBDD representation has $\Omega(1.09^n)$ vertices.

Besides those special functions, in general, we have the following theorem.

Theorem 4 [11] The fraction of Boolean functions whose reduced OBDD size with respect to the standard variable ordering differs more than $O(2^{2n/3})$ from *S(n)* is bounded by $O(2^{-n/3})$, where $S(n) = \sum_{0 \le i \le n-1} 2^{2^{n-i}} (1 - (1 - 2^{-2^{n-i}})^{2^i})$, i.e., almost all functions have exponential size OBDDs.

§6.2 Free BDDs

In the last section, we showed that almost all general random functions have exponential size OBDDs. Therefore OBDDs are not efficient enough in general cases. In this section, we introduce one more new data structure developed in the literature.

The following FBDD is a generalization of OBDD, which is more efficient than OBDD.

Definition 6.1 [12] A Binary Decision Diagram (BDD) over a set $L_n = \{x_1, \cdots, x_n\}$ of Boolean variables is a directed acyclic graph with one source and at most two sinks labeled by 0 and 1. Each non-sink node v is labeled by a Boolean variable $l_v \in L_n$, and has two outgoing edges, one labeled by 0 and the other labeled by 1. The *computation path* for an input $a = (a_1, \cdots, a_n)$ starts at the source. At an inner node with label x_i, the outgoing edge with label a_i is chosen. The BDD P represents a Boolean function f if the computation path for each input a leads to the sink with label $f(a)$.

A BDD is called a *Free Binary Decision Diagram* (FBDD) if, on each path, each variable is tested at most once. An OBDD is an FBDD with the property that on each path the variables are tested in a fixed order.

Example 6.4 Figure 6.1 shows the FBDD and OBDD of the function

$$f(x_1, x_2, x_3, x_4) = \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2x_4 + x_1\bar{x}_3x_4 + x_1x_2x_3.$$

Compared with OBDDs, FBDDs provide more freedom of variable ordering, and therefore result in more compact representations. The FBDD of the function F_4 in Example 6.1 is shown in Figure 6.2. The FBDD works well for Example 6.1. The HSB function also has a compact FBDD representation, which is not shown in the thesis. It is not known whether or not the multiplier function has a compact FBDD.

Even for those hard examples, FBDD are efficient. In general, we have the following result about the complexity of FBDDs.

Theorem ([11]) For a random $n \in \{2^l, \cdots, 2^{l+1} - 1\}$, the probability that the following property P holds tends to 1 as $l \to \infty$.

P: the fraction of Boolean functions whose minimal read-once branching program size is less than $W(n)(1-o(1))$ is bounded by $O(2^{-n/3+\delta n})$ for an arbitrary $\delta > 0$; where $W(n)=\sum_{i=1}^{n} W(i,n)$, $W(i,n)=\min\{2^{i-1}, 2^{2^{n-i+1}}\}$.

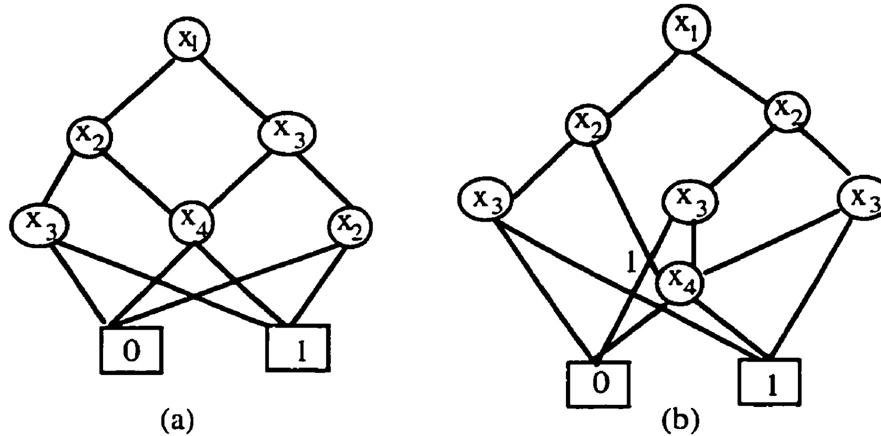


Figure 6.1 Example of an FBDD and an OBDD for the same function *f*.

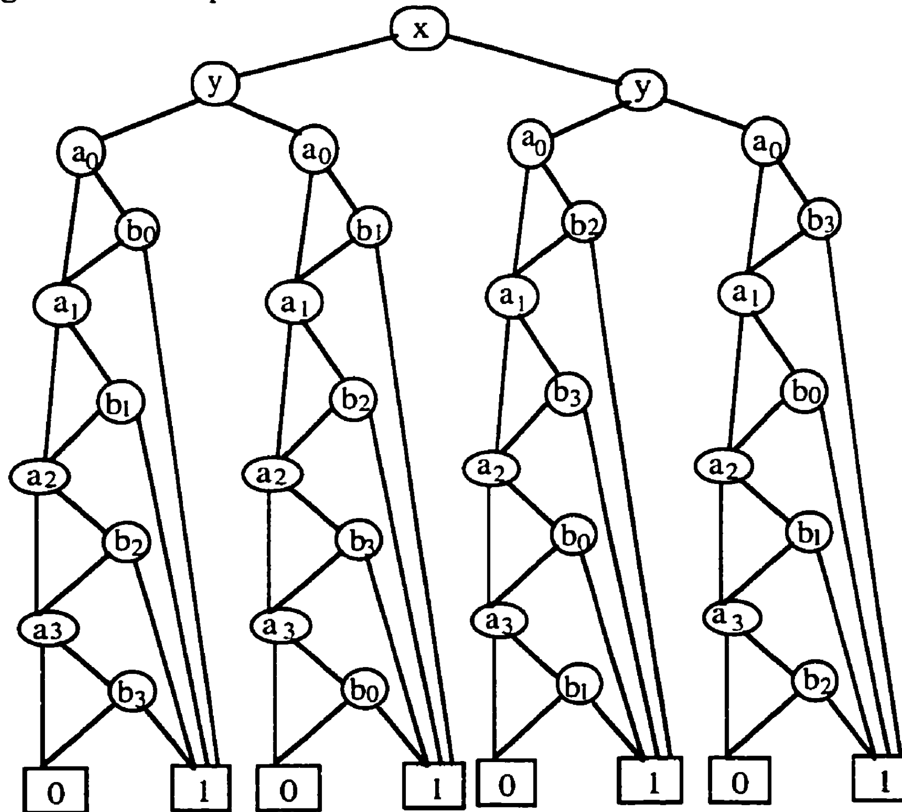


Figure 6.2 FBDD of F_4 in Example 6.1

Therefore in general cases, the FBDDs still have exponential size for almost all Boolean functions. However, no function has been proven to have exponential size FBDD. In the

next chapter, we will show some hard examples which have exponential size FBDDs. We also propose some new data structures for Boolean functions. Those hard examples for FBDD have constant size representation in the new data structure.

Chapter 7[3] A Unified Data Structure

In Chapter 6, we presented some hard example functions which OBDDs fail to represent in compact forms. Graphic data structures other than OBDDs were also presented. In this chapter, we propose new data structures for Boolean functions called D-lists. D-lists provide a unified framework for previous data structures. Product terms and OBDD, though different in representation, are special cases of D-lists. The primary result about the efficiency of SOP or BDD, and the D-lists is that if a Boolean function has polynomial size SOP form or BDDs, then it has polynomial size D-list representation. However, on the other hand, we find examples with exponential size SOPs and BDDs while having constant size D-lists.

D-lists are special integer lists. In the literature, discrete domain problems have been converted into binary domain problems [44] [45]. However the result in this chapter shows that the contrary approach may work better. The true limitation of the traditional data structures and the power of the list representation are clearly shown in this chapter.

The data structures for integer list representation developed in this chapter is closely related to data compression techniques [49]. Actually, almost all non-statistical techniques in [49] are special cases of the data structures developed in this chapter.

---

[3] This chapter is based on the paper "Integer lists - a unified data structures for Boolean functions", which has been submitted to IEEE. Trans. on Computers.

§7.1 Introduction

From what we have presented in this thesis, we can see that the representation of Boolean functions and discrete functions is a hard problem. Up to now, we have introduced many data structures for Boolean functions such as two-level logic expressions (SOP and POS forms), multi-level logic expressions (factored forms), and Reed-Muller expansion; Ordered Binary Decision Diagrams (OBDD), and FBDDs. However, there are still many functions that can not be represented efficiently by any of those data structures.

In this chapter, we present some new and dramatically different data structures. In the literature, discrete functions are translated into Boolean functions and represented by Boolean function data structures [45]. In this chapter, we take the opposite approach. Boolean functions are translated into discrete domains and treated as integer sets and lists.

Many data structures for integer lists are developed in this chapter. To represent an integer list, in order to avoid to enumerate all the integers in the list, the most important method is to identify identical sublists and avoid multiple occurrences of such identical sublists. This purpose can be achieved by special operators and by graphic data structures. However, a monotonic list does not have any identical sublist. We use a new list, called the *first order difference list* which is obtained by taken the difference between adjacent integers in the original monotonic list, to represent the original monotonic list. Such difference lists have well-defined canonical forms. Boolean operations between difference lists are also well-defined. If the first order difference list L is again a monotonic list, then we can take the difference list of L, which is called the *second order difference list* of the original list, to represent the original list.

Boolean functions are treated as monotonic integer lists or the concatenation of monotonic integer lists and therefore can be represented by difference lists. Product terms and OBDDs are all special representation of the first order difference lists. If a function has polynomial size BDDs or SOPs, then it has a polynomial size first-order difference list. On

the other hand, example functions are found which have exponential size BDDs and SOPs but constant size second-order difference lists. This unification of data structures not only opens up many new ways to represent Boolean functions but also points out the limitation of traditional data structures. For functions with monotonic first order difference lists, BDDs and SOP forms fail to represent them in any compact way.

The rest of this chapter is organized as follows. In Section 7.2, we study integer list representations in detail. In Section 7.3, we study the integer list representation of Boolean functions. It is informally shown in this section that product terms and OBDDs all correspond to some special difference lists. In Section 7.4, we show some examples which require exponential size BDDs and SOP forms while they have constant size second order difference lists. In Section 7.5, we develop the operation rules for Boolean functions represented in difference lists. Finally in Section 7.6, we conclude the chapter.

§7.2 Integer Lists and Their Representations

In this section, we introduce various data structures for integer lists.

§7.2.1 Integer Lists

Definition 7.1 Let N denote all natural numbers, an *integer list* L *of length* n is a map $L:[1,n] \rightarrow N$, such that for all $i \in [1,n]$, $L(i) \in N$. In this chapter, integers are written in both binary and numerical form.

The *i-th* element of L is $L(i)$. The integer list L is denoted as $\{L(1), L(2),..., L(n)\}$, where n is the length of the list. The first element $L(1)$ is the *head*, the list $\{L(2),..., L(n)\}$ is the *tail*. The empty list is denoted by $\varnothing$. The list L is bounded by the interval $[min(L), max(L)]$, where $min(L)$ is the minimum element of L and is called the *lower bound*, $max(L)$ is the maximum element of L and is called the *upper bound* of L. $\sum_{i=1}^{n} L(i)$ is called the *sum* of the list L. The number $max(L) - min(L)$ is the *distance* of the list L.

There are some special lists.

Definition 7.2 If a list L satisfies the condition that $L(1)=L(2)=\ldots=L(n)$, where n is the length of the list L, then L is a *flat list*. If every element in a list occurs only once in the list, then this list is a *simple* list. A list L is *monotonic* if the head $L(1)$ is the minimum integer of L, and the tail T is also a monotonic list. The empty list is a monotonic list. A monotonic list is a simple list.

A list of length 1 is called a *single-term* list. Single-term lists are flat and monotonic.

Example 7.1 The list {1, 1, 1} is a flat list. The list {1, 2, 4, 6} is a monotonic list.

We mainly consider the set of lists L which satisfy the condition $n_0 \le \min(L) \le \max(L) \le 2^n - 1 + n_0$, where n and n_0 are some integers. This set is denoted by Δ. Among them, the set of monotonic lists is denoted by Λ.

Some operations can be defined in Δ. The most important operation is the concatenation. One integer list can be appended to another to form a new list. The set Δ is closed under concatenation of lists.

Definition 7.3 The concatenation (@) of two lists L_1 and L_2 is defined as

$$@: \Delta \times \Delta \to \Delta$$
$$@(L_1, L_2) = \{L_1(1), \cdots, L_1(n_1), L_2(1), \cdots, L_2(n_2)\}$$

where n_1 (n_2) is the length of L_1 (L_2). Normally, $@(L_1, L_2)$ is written as $L_1 @ L_2$. Under this notation, an integer list L can be represented as $\{L(1)\} @ \{L(2), \cdots, L(n)\}$. It is easy to define the concatenation of several lists as well.

Definition 7.4 A list is *piecewise flat* if it is the concatenation of several flat lists. A list is *piecewise monotonic* if it is the concatenation of several monotonic list. A monotonic list can be viewed as a piecewise monotonic list as well.

A *non-trivial* piecewise flat list is a the concatenation of several flat lists which are not all single-term lists. A *non-trivial* piecewise monotonic list is the concatenation of several monotonic lists which are not all single-term lists. A list is a *non-monotonic* list if it is not piecewise monotonic. The set of all piecewise monotonic lists is denoted by Λ^*.

Another operation of integer lists is the separation operation. While the concatenation combines two or more lists into one, the separation operation separates one list into several lists.

Definition 7.5 The separation ($\overline{@}$) of a list is the operation that separates the list $L_1 @ L_2$ into two lists L_1 and L_2. The formal definition is as follows.

$$\overline{@}: \Delta \rightarrow \Delta \times \Delta$$

$$\overline{@}(L_1 @ L_2) = (L_1, L_2)$$

We denote the pair $(L_1, L_2) = L_1 \perp L_2$. Based on the separation of one list into two lists, the separation of one list into several lists can be defined as well. We denote $\Delta^k = \overbrace{\Delta \times \cdots \times \Delta}^{k}$ and define $L_1 \perp L_2 \perp \cdots \perp L_k$ similarly as for $L_1 \perp L_2$. $L_1 \perp L_2 \perp \cdots \perp L_k$ is called a *multiple list.*

A list can be separated into a multiple list; on the other hand, several lists can be combined to form one list using the concatenation operator. The reason for doing those operations is to find the best representation for the lists.

§7.2.2 The Representation of Single Non-monotonic Lists

The simplest way to represent a list is to enumerate all the element in the list as $\{L(1), L(2), \ldots, L(n)\}$. However such notation often results in long list and therefore impractical. In the following, we study the representation of lists. The representation of non-monotonic lists is different from the representation of monotonic lists. To represent a non-monotonic list, we mainly decompose a list into several sublists so that some of them are identical or reverse lists, then we introduce new notations to represent such relationships, which result in short representations.

Definition 7.6 Two lists of the same length are *identical* if their head are the same integer; moreover, their tails are identical as well. Two identical lists are denoted as $L_1 = L_2$. For a list $L_1 = \{m_1, \cdots, m_k\}$, the list $L_2 = \{m_k, \cdots, m_1\}$ is called the *inverse list* of L_1, i.e., $L_2 = \overline{L_1}$. For two lists L_1 and L_2 of length n_1 and n_2 respectively, where $n_1 \le n_2$, if

there is an index i such that for index $i+1 \le j \le i+n_1$, $L_2(j) = L_1(j-i)$, then the list L_1 is a *sublist* of L_2.

Definition 7.7 For two lists L_1 and L_2, the list $L_1 @ L_2 @ \overline{L_1}$ is a *reflective* list with respect to L_2. This list is denoted by $< L_1, L_2 >$. In the case $L_2 = \varnothing$, the list $L_1 @ \varnothing @ \overline{L_1}$ is shorten as $< L_1 >$ and is called the *reflective list* of L_1.

In the case $L_2 = \{x\}$ has only one element x, the list $L_1 @ \{x\} @ \overline{L_1}$ can be shorten as $< L_1, x >$, and the list is called the reflective list of L_1 with respect to x.

Example 7.2 Consider the list $D = \{1,\ 11,\ 1,\ 1011,\ 1,\ 11,\ 1\}$. Using the reflective list, it can be represented as $<\{1, 11, 1\}, 1011> = <<<1>, 11>, 1011>$. Therefore $D = <<<1>, 11>, 1011>$. The list $<1, 2, 3, 1, 3, 2, 1>$ can be denoted as $<\{1,2,3\}, 1>$.

Besides reflective lists, we can also introduce new notations for identical sublists. Among identical sublists, flat lists are of particular importance. A flat list has only one distinct element. A non-flat list can be decomposed into the concatenation of several flat sublists. In the following, we focus on the representation of lists based on the decomposition of identical sublists. Traditional data structures for Boolean functions are mainly related to such representations.

Definition 7.8 For a list of the form $D_1 @ D_2$, if $D_1 = D_2$, then we denote $D_1 @ D_2 = 2 * D_1$. Similarly we can define the notation $n * D_1$ to represent the concatenation of n-copies of the list D_1. As a special case, the flat list of length n can be represented as $n*\{s\}$, where s is the only integer in the list. The operator * is called the *counter operator*.

Proposition 7.1 Every list L can be represented as the concatenation of several flat lists, i.e., $L = L_1 @ L_2 @ \cdots @ L_k$ such that each L_i is a flat list.

Definition 7.9 If a list is represented by the concatenation @ and the counter operator * of sublists, then this list is said in *I-form*. If the * operator is only used of the form $n*\{s\}$, then the list is said in the *F-form*.

Example 7.3 The list $L = \{2,\ 1,\ 1,\ 1,\ 2\}$ can be represented as $\{2\}@\{1,1,1\}@\{2\}$, which in turn can be denoted as $\{2\}@3*\{1\}@\{2\}$. $\{2\}@3*\{1\}@\{2\}$ is the F-form of L.

A list may have many I-form representation. The following is such an example.

Example 7.4 The list *L={* 2, 3, 3, 3, 4, 2, 3, 3, 3, 4, 5, 2, 3, 3, 3, 4, 5}=2*{2, 3, 3, 3, 4}@{5, 2, 3, 3, 3, 4, 5}=2*({2}@3*{3}@{4})@{5,2}@3*{3}@{4, 5}={2, 3, 3, 3, 4}@2*{2, 3, 3, 3, 4, 5}.

For such list, we need to define some canonical forms and minimization procedures. It turns out that there are many possible canonical forms for a list in I-form.

Definition 7.10 The *first level canonical form* of a list in I-form is the form that every flat sublist is written in the * operator, i.e., its F-form.

Example 7.5 For the list in above example *L={* 2, 3, 3, 3, 4, 2, 3, 3, 3, 4, 5, 2, 3, 3, 3, 4, 5}, the first level canonical form is as *L=* {2}@3*{3}@{4}@{2}@3*{3}@{4}@{5}@{2}@3*{3}@{4}@{ 5}.

Definition 7.11 The *second level canonical form* of a list in I-form is obtained from the first level canonical form by replacing the largest identical sublist by * operator, starting from the beginning of the list.

Example 7.6 The second level canonical form of *L=* {2}@3*{3}@{4}@{2}@3*{3}@{4}@{5}@{2}@3*{3}@{4}@{5} is 2*({2}@3*{3}@{4})@{5}@{2}@3*{3}@{4}@{5}.

We can continue to define such canonical forms. The more we reduce, the fewer number of integers we need to use to represent the list. This process will eventually stop because the list is of finite length.

Besides the above defined canonical forms, we can also define graphic structures to represent a list. This is of particular interest because of the popularity of the OBDD, which is a graphic data structure.

Definition 7.12 A *binary graph* is a rooted directed graph with vertex set V containing two types of vertices. A non-terminal vertex v has two children $l(v)$, $r(v)$. The edges $(v, r(v))$ and $(v, l(v))$ are labeled by integers. A terminal vertex v has no children and is labeled by an integer list.

For the node v_0 with two children v_1 and v_2, each child represents an integer list, and the integer list represented by v_0 is defined by $L_{v_0} = (e_1 * L_{v_1}) @ (e_2 * L_{v_2})$, where e_1 and e_2 are the integer labels of the edge (v_0, v_1) and (v_0, v_2) respectively. The list represented by a terminal node is the integer list label of the node.

Example 7.7 The following is a binary graph representation of the list 3*(3*{4} @4*{1})@16*{1}. The edges without integer labels shown in the figure are labeled by 1. For a node *v*, the left child is the *l(v)*, and the right child is the *r(v)*.

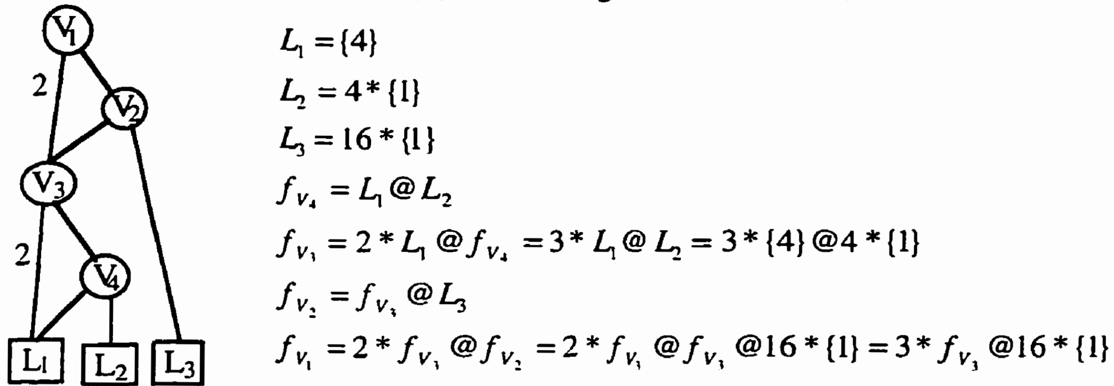


Figure 7.1 Binary Graph Representation of an Integer List

When the labeling of the edges are restricted to the form 2^i, the binary graph corresponds to the OBDD data structure for Boolean functions. This correspondence will be shown in later section. However, we can remove the 2^i restriction and have edges labeled by any integers. The following Figure 7.2 shows another binary graph representing the same list in above example while the edges are labeled differently.

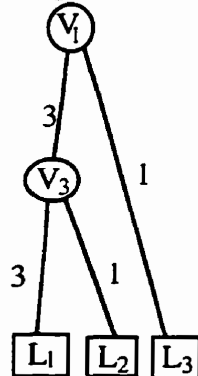


Figure 7.2 Another Binary Graph of an Integer List

Similarly we can define graphs with multiple children rather than restricted to two children to represent integer lists. Therefore among list representation graphs, OBDDs correspond to a very restricted class of graphs. This understanding can help us define new graphic structures for Boolean functions. [33].

§7.2.3 The Representation of Multiple Non-monotonic Lists

For multiple non-monotonic list of the form $L_1 \perp L_2 \perp \cdots \perp L_k$, we develop one more notation.

Definition 7.13 We define the notation $L_1 \hat{\perp} L_2$, where $L_1 = \{L_1(1), \cdots, L_1(n_1)\}$,

$$L_1 \hat{\perp} L_2 = (L_1(1) @ L_2) \perp (L_1(2) @ L_2) \perp \cdots \perp (L_1(n_1) @ L_2).$$

Example 7.8 $\{1,2,3,4\} \perp \{5,2,3,4\} \perp \{6,2,3,4\} = \{1,5,6\} \hat{\perp} \{2,3,4\}$.

§7.2.4 Representation of Piecewise Monotonic Lists

The above techniques can reduce the length of the representation for non-monotonic lists. For a monotonic list, the above techniques do not work at all. For example, when a monotonic list is in the F-form, every sublist has to be a single-term list. We need new ways to represent non-trivial piecewise monotonic lists. In this section, we introduce ways to represent piecewise monotonic lists. We first look at the monotonic lists. Recall Δ is the set of lists L satisfying $n_0 \le \min(L) \le \max(L) \le 2^n - 1 + n_0$; $\Lambda \subset \Delta$ is the set of monotonic lists.

Definition 7.14 The operator $D: \Lambda \to \Delta$ is defined as: for a monotonic list $L = \{m_1, m_2, \cdots, m_n\}$, $D(L) = \{m_1, m_2 - m_1, m_3 - m_2, \cdots, m_n - m_{n-1}\}$. The list *D(L)* is called the *first order difference list* (D-list) of *L*. The operator D is called the *difference operator.*

Example 7.9 The list *L*={13, 14, 15, 16} has *D(L)*={13, 1, 1, 1}={13}@ 3*{1}.

For a monotonic list, its difference list is just a general integer list. On the other hand, general integer list can be seen as a difference list of a monotonic list. The corresponding

monotonic list is generated by the following proposition.

Proposition 7.2 If a monotonic list L has the integer list $M=\{m_0,d_1,d_2,\cdots,d_n\}$ as its difference list, then L is of the form: $L=\{m_0,m_0+d_1,m_0+d_1+d_2,\cdots,m_0+\sum_{i=1}^{n}d_i\}$.

Definition 7.15 The correspondence from $M=\{m_0,d_1,d_2,\cdots,d_n\}$ to $L=\{m_0,m_0+d_1,m_0+d_1+d_2,\cdots,m_0+\sum_{i=1}^{n}d_i\}$ is denoted by $\overline{D}$, i.e., $\overline{D}:\Delta\rightarrow\Lambda$ is defined as $\overline{D}(M)=L$. Therefore monotonic lists can be represented by their difference lists.

For a monotonic list, its difference list may not be monotonic anymore. Therefore the techniques for non-monotonic lists can be used to represent the difference list.

Difference lists can be extended into piecewise monotonic lists. Therefore piecewise-monotonic lists can be represented by their difference lists as well.

Definition 7.16 For a piecewise monotonic list, its *difference list* is a multiple list obtained by the operator $D^*:\Lambda^*\rightarrow\Delta^k$ defined as $D^*(M_1@\cdots@M_k)=D(M_1)\perp\cdots\perp D(M_k)$.

Example 7.10 The integer list L={36, 37, 38, 39, 44, 45, 46, 47, 52, 53, 54, 55, 60, 61, 62, 63} is a piecewise monotonic since L = {36, 37, 38, 39}@{44, 45, 46, 47} @{52, 53, 54, 55,} @{60, 61, 62, 63}. The list $D(L)$= {36}@3*{1}⊥44@3*{1}⊥52@ 3*{1}⊥60@3*{1}={36, 44, 52, 60}$\bar{\perp}$3*{1}.

In the case that the difference list of a monotonic list is still a monotonic list, then the difference list representation is still not enough since it can not use any technique for the representation of the non-monotonic list at all. In this case, we need to apply the difference operator recursively until we get a non-monotonic difference list. More formally, we have the following definition.

Definition 7.17 For the difference operator $D:\Lambda\rightarrow\Delta$, if a list L has $D(L)\in\Lambda$, we can apply the operator D to $D(L)$ again. The result is denoted as $D^2(L)$. $D^2(L)$ is the *second order difference list* of L.

Example 7.11 Consider the list L_k={1, 1+2, 1+2+3, ..., $\sum_{i=1}^{k}i$}. D(L_k)={1, 2, ..., k} which is still a monotonic list. Therefore we can represent L_k by $D^2(L_k)$=(k-1)*{1}. If

$k = 2^n$, then $D^2(L_k)$ is of constant size while L_k and D(L_k) are of exponential size. In later section, we will prove that for functions similar to L_k, its OBDD and FBDD are both of exponential size. Therefore the power of the difference list over OBDD or FBDD is clearly shown.

For piecewise monotonic lists, if we have the form $D^*(L) = L_1 \hat{\perp} L_2$, if either L_1 or L_2 is a monotonic list again, we can apply the D operator again. The following is such an example.

Example 7.12 Consider the list in the example $L = \{36, 44, 52, 60\} \hat{\perp} 3 * \{1\}$. If we apply the difference operator D to the list {36, 44, 52, 60} again, then we get a list representation $D(L) = (\{36\} @ 3 * \{8\}) \hat{\perp} 3 * \{1\}$.

In summary, piecewise monotonic lists can be represented by their difference lists, either first order difference lists or higher order difference lists, which generally results in more compact representation.

§7.3 Integer List Representation of Boolean Functions

In this section, we establish the relationship between Boolean functions and integer lists.

§7.3.1 Integer Lists and Boolean Functions

Definition 7.18 An *n_0-based integer map* is defined as a map $I: B^n = \{(x_1, \cdots, x_n) | x_i = 0,1\} \rightarrow [n_0, 2^n - 1 + n_0]$ by the rule $I(x_1, \cdots, x_n) = \sum_{j=0}^{n-1} 2^j x_{i_j} + n_0$, where $(x_{i_1}, \cdots, x_{i_n})$ is a permutation of the variable $(x_1, \cdots, x_n)$, n_0 is an integer.

The *straight map* $I_0(x_1, \cdots, x_n) = \sum_{i=1}^{n} 2^{n-i} x_i$ maps a minterm $(x_1, \cdots, x_n)$ to the integer $x_1 \cdots x_n$ (binary form). The integer map $I_1(x_1, \cdots, x_n) = \sum_{i=1}^{n} 2^{n-i} x_i + 1$ is the *standard map*, which maps a minterm $(x_1, \cdots, x_n)$ to the integer $x_1 \cdots x_n + 1$. The following Figure 7.3

shows the correspondence between the set of minterms into integer sets under straight and standard map.

000	0	1
001	1	2
010	2	3
011	3	4
100	4	5
101	5	6
110	6	7
111	7	8

Figure 7.3 Straight Map and Standard Map

Recall Δ is the set of all integer lists bounded by $[n_0, 2^n - 1 + n_0]$, $\Lambda^* \subset \Delta$ is the subset of piecewise monotonic lists, and *F(n)* the set of *n*-variable Boolean functions.

Definition 7.19 Let Γ denote the set of all subsets of the interval $[n_0, 2^n - 1 + n_0]$. Given an integer map $I: B^n = \{(x_1, \cdots, x_n) | x_i = 0,1\} \rightarrow [n_0, 2^n - 1 + n_0]$, I induces a map $I: F(n) \rightarrow \Gamma$ which is defined as to map a minterm set into its corresponding integer set. The induced map $I: F(n) \rightarrow \Gamma$ is 1-1 and onto. For a function $f \subset B^n$, *I(f)* is the *associated integer set of f.*

Once an integer map is given, we do not need to distinguish between Boolean functions and integer sets. Integer sets are represented by integer lists. Integer lists are different from integer sets. The ordering of elements in an integer set is not important. Moreover, every element can appear once in an integer set. However, there is a well defined map from the set Δ to the set Γ.

Definition 7.20 The map $S: \Delta \rightarrow \Gamma$ is defined as: for an integer list *L*, its *associated integer set S(L)* is the set consists of integers in the list. Two integer lists which have the same associated integer set are said to be *equivalent.*

The combination of S and I maps each integer list to a Boolean function as follows: $\Delta \xrightarrow{S} \Gamma \xleftrightarrow{I} F(n)$ Therefore each list represents a Boolean function. On the other hand, a Boolean function can be represented by many different integer lists. We will use some special integer lists, mainly monotonic and piecewise monotonic lists, to

represent Boolean functions. The reason for doing so is that monotonic and piecewise monotonic lists are easy to manipulate.

§7.3.2 Default Representation of Boolean Functions

When monotonic lists are used to represent Boolean functions, we call it the *default representation.*

Definition 7.21 Given an integer map *I*, there is a well-defined map $I^*: F(n) \to \Lambda$. For a function f, $I^*(f)$ is the sorted (monotonic) list of *I(f)*. $I^*(f)$ is called the *default list* of *f*. The minimum integer of *I(f)* is denoted by *min(f)*, and the maximum integer of *I(f)* is denoted by *max(f)*.

Example 7.13 Consider a Boolean function $f(x_1, x_2, x_3, x_4)$ defined by the minterm set {1100, 1101, 1110, 1111}, where 1100 represents the minterm $x_1 x_2 \bar{x}_3 \bar{x}_4$, similar rule holds for other minterms. It is easy to see that $f(x_1, x_2, x_3, x_4) = x_1 x_2$. Under the standard map, $f(x_1, x_2, x_3, x_4)$ has the default list as {13, 14, 15, 16}.

Definition 7.22 Given an integer map *I*, the default list $I^*(f)$ is a monotonic list. Combining with the map $D: \Lambda \to \Delta$, we have a map $DI^*: F(n) \to \Delta$ defined as $DI^*(f) = D(I^*(f)) \in \Delta$. The list $D(I^*(f))$ is called the *difference list* of Boolean function *f*. The reader is reminder that the difference lists defined here is totally different from the difference lists used in Prolog as a programming technique.

When Boolean functions are represented by their difference lists, it is called the difference list (D-list) representation of Boolean functions.

Different integer maps can result in different D-list for Boolean functions. Consider the following example.

Example 7.14 The function $x_1 x_2 + x_3 x_4$ represents the set of minterms {0011, 0111, 1011, 1100, 1101, 1110, 1111}. Its difference list can be reduced as {3}@2*{4}@4*{1} if the straight map is used. However, if we use the standard map, then the function has the D-list as 3*{4}@4*{1}, which is simpler than the straight map based D-list. From now

on, we assume the integer map is the standard map.

Difference list representation of Boolean functions is very powerful. Product terms have special difference lists. OBDDs and other BDDs can be viewed as graphic representation of first order difference lists as well. Therefore if a function has polynomial size SOP form or BDDs, then it has polynomial size D-list representation.

Proposition 7.3 In an n-dimensional Boolean space $B^n = \{x_1, \cdots, x_n\}$, the product term $x_1 \cdots x_k$ has difference list as D($x_1 \cdots x_k$)={ $x_1 \cdots x_k 0 \cdots 0+1$ }$@(2^{n-k}-1)*\{1\}$, i.e., its D-list is mainly a flat list.

However, flat difference lists are more general than product terms $x_1 \cdots x_k$. If the length of the flat D-list is not of the form $(2^{n-k}-1)$, then the corresponding function can not be represented by a product term.

Example 7.15 Consider the function with the D-list {2}@6*{1}, in traditional sum of product form, this function is $x_1 + x_2 + x_3 + x_4$.

General product terms also have regular difference lists, the size of such difference lists are bounded by the number of variables.

Example 7.16 The product term $x_1 --- x_5 ---$ has the D-list $\{137\}@(2^3-1)*((2^3-1)*\{1\}@\{9\})@(2^3-1)*\{1\}$. In general, for $2i$ variables, the product term $x_1 - \cdots - x_{i+1} - \cdots -$ can be represented as $\{10 \cdots 010 \cdots 0\}@(2^{i-1}-1)*((2^{i-1}-1)*\{1\}@\{2^i+1\})@(2^{i-1}-1)*\{1\}$.

Reflective lists can also be used to represent general product terms.

Example 7.17 Consider the D-list D={101010+1}@{1, 11, 1, 1011, 1, 11, 1}. Its corresponding function is the product term $x_1 - x_3 - x_5 -$. Since{1, 11, 1, 1011, 1, 11, 1}=<<<1>, 11>, 1011>, therefore D= {101010+1}@ <<<1>, 11>, 1011>.

It can be proven that the D-list of a product term is a reflective list. However, reflective lists are more general than product terms. There are reflective lists that can not be represented by a product term.

Example 7.18 Consider the odd-even parity function of four variables L={0001,

0010, 0100, 0111, 1000, 1011, 1101, 1110}. The D-list *D(L)* is {2}@{1, 2, 3, 1, 3, 2, 1}={2}@ <{1,2,3}, 1>.

The OBDD of a Boolean function corresponds to the binary graph representation of the D-list of the function. In the following, we show such correspondence by examples.

Example 7.19 Consider functions $x_1x_4 + x_2x_5 + x_3x_6$ and $x_1x_2 + x_3x_4 + x_5x_6$. Their difference lists are of the form: {1001+1}@3*{2}@2*{3, 1}@2*{2, 1, 1}@{5, 1, 1, 1}@2*{2}@4*{1}@{3, 1}@4*{1}@{2, 1, 1}@4*{1} and 3*(3*{4}@4*{1}) @16 {1}. Represented by binary graphs of lists, these two lists have their binary graph as shown in Figure (I) and Figure (II) of Figure 7.4, where edges are labeled by 1 if there is no other label shown in the graph for the edge. For a vertex *v*, the left child is the *l(v)* and the right child is the *r(v)*.

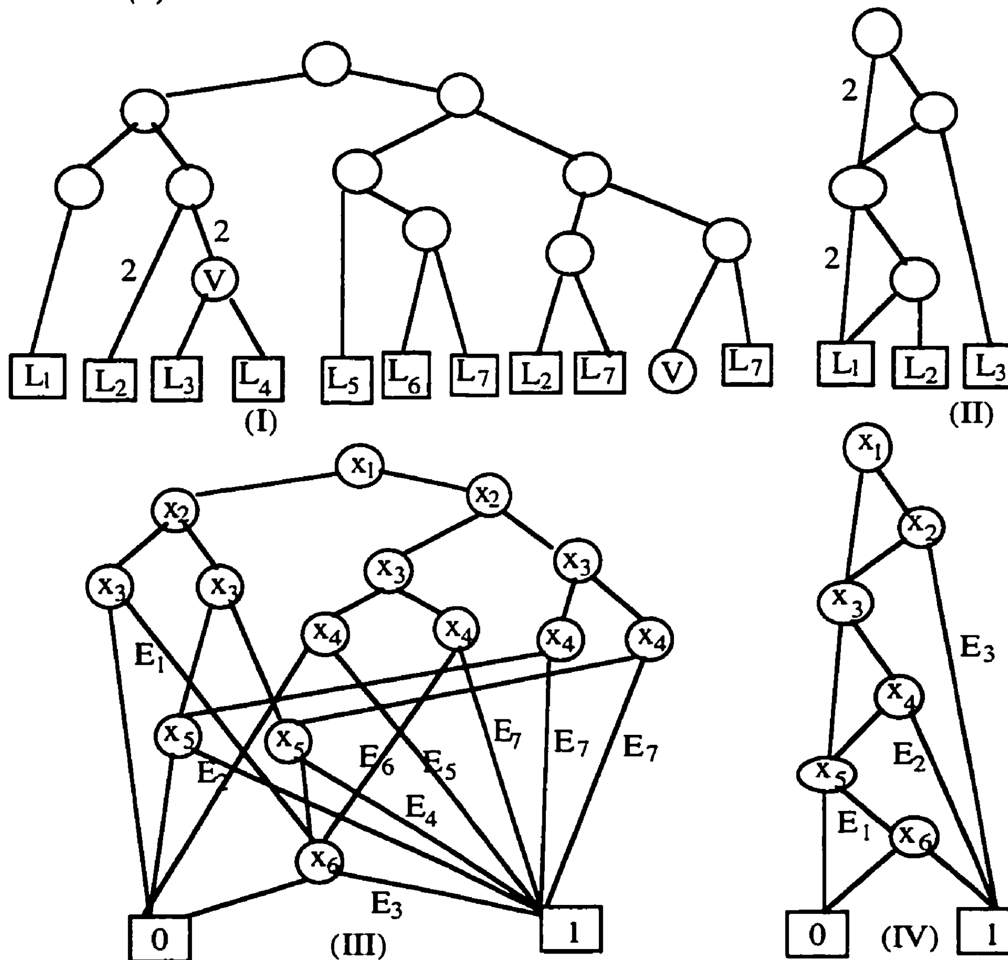


Figure 7.4 OBDDs and Binary Graphs of D-lists

In Figure (I), $L_1 = \{1001,2,2,2\}; L_2 = \{3,1\}; L_3 = \{2\}; L_4 = \{1,1\};$ $L_5 = \{5,1,1,1\}; L_6 = \{2,2\}; L_7 = \{1,1,1,1\}$. In Figure (II), $L_1 = \{4\}; L_2 = 4*\{1\}; L_3 = 16*\{1\}$. The Figure (III) and Figure (IV) show the corresponding OBDDs.

It is not hard intuitively to see the correspondence between OBDDs and the binary graphs of the difference lists. The complete rule of the correspondence between OBDDs and the binary graphs of the D-lists of Boolean functions is lengthy to state. Therefore we leave this material out. However, from the correspondence of the figures, we can still see some simple rules which would be true in general cases.

Vertices in an OBDD are translated into different objects in the corresponding binary graph of the D-list. If the vertex v_0 in the OBDD has two non-terminal children v_1 and v_2, then the vertex remains in the binary graph, moreover, the edge (v_0, v_1) is labeled by the integer 2^i, where the vertex v_0 is labeled by x_{i_0} and v_1 is labeled by x_{i_0+i+1}. For example, the edge (x_1, x_3) is labeled by 2^1=2. If a vertex v_0 in the OBDD has two children as terminal vertices, then this vertex is replaced by a terminal vertex in the binary graph. If a vertex v_0 in the OBDD has one child v_1 as the 1 terminal vertex, then we add one terminal vertex in the binary graph to represent an integer list. This terminal integer list is determined by the labeling of the vertex v_0 and the edge (v_0, v_1). In the Figures, the list L_i corresponds to the edges E_i. This correspondence shows that if a function has polynomial size OBDD, then it has polynomial size D-list representation.

§7.3.3 Piecewise Monotonic Representation of Boolean Functions

While the difference list representation of Boolean functions is powerful, it has it own problems. One problem is that the number of redundant variables can affect the representation of the same product term.

Example 7.20 The function $x_1x_2 + x_3x_4$ has the default list {0011, 0111, 1011, 1100, 1101, 1110, 1111}. Its difference list is 3*{4}@4*{1} if the function is in B^4.

However, if the function is in B^6, then the same function has a very different difference list {13}@3*{1}@{13}@3*{1}@{13}@20*{1}=3*({13}@3*{1})@17* {1}.

To overcome this problem, we use piecewise monotonic lists and their difference lists to represent Boolean functions instead of default lists.

Example 7.21 The integer set of the product term $1--1--$ can be represented by the piecewise monotonic list {37, 38, 39, 40}@ {45, 46, 47, 48 }@{53, 54, 55, 56}@{61, 62, 63, 64}. By definition, the difference list of this piecewise monotonic list is L={37,1,1,1}⊥{45,1,1,1}⊥{53,1,1,1}⊥{61,1,1,1}={37}@3*{1}⊥{45}@3*{1}⊥{53}@3*{1} ⊥{61}@3*{1}={37,45,53,61}$\hat{\perp}$3*{1}. If we apply the difference operator recursively, we will have $D(L)$= ({37}@3*{8})$\hat{\perp}$3*{1}.

In general, for $2i$ variables, the product term $x_1-\cdots-x_{i+1}-\cdots-$ has the recursive D-list as $(a@(2^{i-2}-1)*\{2^{i+1}\})\hat{\perp}(2^{i-2}-1)*\{1\}$, where a is an integer. When the number of variables change, the only change need to make is the count i, therefore, this representation is close to the cubical representation of the product term.

In summary, a Boolean function can be represented by the difference list of the default list of the function, or by the difference list of a piecewise monotonic list of the function. When the difference list is written in I-form, it is a generalization of the product term structure. When the difference list is represented by binary graphs, then it corresponds to the OBDD of the function.

§7.4 The Power of Difference Lists

In the last section, we have shown that SOPs and OBDDs are special cases of first order difference lists. Therefore if a function has polynomial size SOPs or OBDDs, then it has a polynomial size difference list representation.

In the literature, example functions are constructed to demonstrate some theory such as in [8]. In this section, we take similar approach. We show examples which require exponential size FBDDs and SOP forms, while their second order D-lists are of constant size. These examples are important because they give concrete insights into the difference lists and show the limitations of data structures such as SOP and BDDs.

We define some Boolean functions by defining their integer sets. We can also refer an integer as a minterm.

$$f(k) = \{0, 1, 1+2, \cdots, \sum_{i=0}^{2^k-1} i\}$$

$$f(k+1) = f(k) \cup \{\sum_{i=0}^{2^k} i, \sum_{i=0}^{2^k} i + 2^k + 1, \sum_{i=0}^{2^k} i + (2^k+1) + (2^k+2), \cdots, \sum_{i=0}^{2^k} i + \sum_{j=1}^{2^k-1} (2^k + j)\}$$

$$g(k) = f(k+1) - f(k) = \{\sum_{i=0}^{2^k} i, \sum_{i=0}^{2^k} i + 2^k + 1, \sum_{i=0}^{2^k} i + (2^k+1) + (2^k+2), \cdots, \sum_{i=0}^{2^k} i + \sum_{j=1}^{2^k-1} (2^k + j)\}$$

$$h(k) = \{\sum_{i=0}^{2^k} i, \sum_{i=0}^{2^k} i + 2^k + 1, \sum_{i=0}^{2^k} i + (2^k+1) + (2^k+2), \cdots, \sum_{i=0}^{2^k} i + \sum_{j=1}^{2^{\lfloor \frac{k}{2} \rfloor}} (2^k + j)\}$$

We have the following equalities.

(1) $\sum_{i=0}^{2^k-1} i = \frac{1}{2} * 2^k * (2^k - 1) = 2^{k-1} * (2^k - 1) = 0\overbrace{01\cdots1}^{k}\overbrace{10\cdots0}^{k}$

(2) $2^{k-1} * (2^k + 1) = 0\overbrace{10\cdots0}^{k}\overbrace{10\cdots0}^{k}$

(3) $\sum_{i=0}^{2^k} i + \sum_{j=1}^{2^k-1} (2^k + j) = \sum_{i=0}^{2^k+(2^k-1)} i = \frac{1}{2} * 2^{k+1} * (2^{k+1} - 1) = 1\overbrace{11\cdots1}^{k}\overbrace{00\cdots0}^{k}$

The minterms in function *g(k)* and *h(k)* have *2k+1* bits. Therefore they can be viewed as Boolean functions with *2k+1* variables. We introduce some propositions of *h(k)* (*g(k)*).

Proposition 7.4 The function *g(k)* contains 2^k minterms. The distance between any two minterms are greater than 1. The last k bit of all minterms are different.

Proof If two minterms n_1 and n_2 have the same last k bit string, then n_1 mod $2^k = n_2$ mod 2^k, i.e., $n_1 - n_2 = m * 2^k$ for some integer m. On the other hand, we have $n_1 = \sum_{j=0}^{i_1} j = \frac{1}{2} i_1 (i_1 + 1)$, $n_2 = \sum_{j=0}^{i_2} j = \frac{1}{2} i_2 (i_2 + 1)$, where $2^k < i_1 < 2^{k+1}$, $2^k < i_2 < 2^{k+1}$, therefore we have $n_1 - n_2 = \frac{1}{2} i_1 (i_1 + 1) - \frac{1}{2} i_2 (i_2 + 1) = \frac{1}{2} (i_1 - i_2)(i_1 + i_2 + 1)$.

If $\frac{1}{2}(i_1 - i_2)(i_1 + i_2 + 1) = m * 2^k$, then $(i_1 - i_2)(i_1 + i_2 + 1) = m * 2^{k+1}$.

Since $(i_1 - i_2) + (i_1 + i_2 + 1) = 2i_1 + 1$, one of $(i_1 - i_2)$ and $(i_1 + i_2 + 1)$ has to be odd, which implies that $2^{k+1} | (i_1 - i_2)$ or $2^{k+1} | (i_1 + i_2 + 1)$. However, $2^{k+1} > (i_1 - i_2)$, $2 * 2^k < i_2 + i_1 < 2 * 2^{k+1}$. Therefore 2^{k+1} can not divide $(i_1 + i_2 + 1)$ or $(i_1 - i_2)$. We arrive at a contradiction. Therefore there are no two minterms with the same last k bit string.

Similarly, if two minterms have distance 1, then $n_1 - n_2 = 2^i$ for some i, which implies that $(i_1 - i_2)(i_1 + i_2 + 1) = 2^{i+1}$. Since $(i_1 - i_2) + (i_1 + i_2 + 1) = 2i_1 + 1$, one of $(i_1 - i_2)$ and $(i_1 + i_2 + 1)$ has to be odd. Suppose $(i_1 + i_2 + 1) = 2i_0 + 1$, then $(i_1 - i_2)(2i_0 + 1) = 2^{i+1}$, which is impossible. Therefore the distance between any two minterms is greater than 1.

Proposition 7.5 For the function *h(k),* there are no two pairs of minterms (n_1, n_2) and (n_3, n_4) in *h(k)* such that $n_1 - n_2 = n_3 - n_4$.

Proof Suppose there are two pairs of minterms (n_1, n_2) and (n_3, n_4) in *h(k)* such that $n_1 - n_2 = n_3 - n_4$. Suppose $n_1 = \sum_{j=0}^{i_1} j$, $n_2 = \sum_{j=0}^{i_2} j$, $n_3 = \sum_{j=0}^{i_3} j$, $n_4 = \sum_{j=0}^{i_4} j$. We can assume that $n_1 > n_2 > n_3 > n_4$, which implies that $i_1 > i_2 > i_3 > i_4$.

This is true because we can always assume $n_1 > n_2$, $n_3 > n_4$, and $n_1 > n_3$. In this case, $n_2 > n_4$, because if $n_2 < n_4$, we will have a contradiction.

We can further assume $n_2 > n_3$. In the case $n_2 < n_3$, we can switch the position of n_2 and n_3, i.e., we will have $n_1 - n_3 = n_2 - n_4$ and $n_1 > n_3 > n_2 > n_4$.

$$n_1 - n_2 = \sum_{j=0}^{i_1} j - \sum_{j=0}^{i_2} j = \sum_{j=i_2+1}^{i_1} j,$$

$$n_3 - n_4 = \sum_{j=0}^{i_3} j - \sum_{j=0}^{i_4} j = \sum_{j=i_4+1}^{i_3} j$$

if $n_2 < n_3$, then $i_2 < i_3$; $\sum_{j=i_2+1}^{i_1} j = \sum_{j=i_4+1}^{i_3} j$ implies that $\sum_{j=i_2+1}^{i_3} j + \sum_{j=i_3+1}^{i_1} j = \sum_{j=i_4+1}^{i_2} j + \sum_{j=i_2+1}^{i_3} j$. Therefore $\sum_{j=i_4+1}^{i_2} j = \sum_{j=i_3+1}^{i_1} j$, i.e., $n_1 - n_3 = n_2 - n_4$, and $n_1 > n_3 > n_2 > n_4$.

Let $i_1 - i_2 = m_1$, $i_3 - i_4 = m_2$. We have $m_1 < m_2$. Moreover,

$$n_3 - n_4 = \sum_{j=i_4+1}^{i_3} j = \sum_{j=i_4+1}^{i_3-m_1} j + \sum_{j=i_3-m_1+1}^{i_3} j, \quad n_1 - n_2 = \sum_{j=i_2+1}^{i_1} j.$$

$$\sum_{j=i_4+1}^{i_3-m_1} j = \sum_{j=i_2+1}^{i_1} j - \sum_{j=i_3-m_1+1}^{i_3} j = \sum_{j=1}^{m_1}(i_2+j) - \sum_{j=1}^{m_1}(i_3-m_1+j) = \sum_{j=1}^{m_1}(i_2+j)-(i_3-m_1+j)$$
$$= \sum_{j=1}^{m_1}(i_1-i_3) = m_1(i_1-i_3)$$

It is always true that $i_4+1>2^k$. Therefore we have $\sum_{j=i_4+1}^{i_3-m_1} j>2^k$, i.e., $m_1(i_1-i_3)> 2^k$. Therefore, $m_1+(i_1-i_3)>\sqrt{2^k}>2^{\lfloor \frac{k}{2} \rfloor}$. On the other hand, $i_1=(i_3-m_1)+m_1+(i_1-i_3)>2^{\lfloor \frac{k}{2} \rfloor}$ $+i_3-m_1 \geq i_4+1+2^{\lfloor \frac{k}{2} \rfloor} \geq 2^k+1+2^{\lfloor \frac{k}{2} \rfloor}$. $n_1=\sum_{j=0}^{i_1} j > \sum_{i=0}^{2^k} i + \sum_{j=1}^{2^{\lfloor \frac{k}{2} \rfloor}} (2^k+j)$. Since $\sum_{i=0}^{2^k} i + \sum_{j=1}^{2^{\lfloor \frac{k}{2} \rfloor}} (2^k+j)$ is the maximum integer in *h(k)*, we arrive at a contradiction. The proposition is proven by contradiction.

Based on the above two propositions, we study the representation of the function *h(k)*.

Proposition 7.6 The size of the SOP form of *g(k)* is 2^k. The size of the SOP form of *h(k)* is $2^{\lfloor \frac{k}{2} \rfloor}$. Every product term of *g(k)* and *h(k)* is a minterm.

Definition 7.23 If a Boolean function *f* does not contain any product term other than minterms, then the function is called a *sparse* function. Functions *g(k)* and *h(k)* are sparse.

Next we study the FBDD of the function *h(k)*.

We are interested in the FBDD with only one sink 1. A non-sink node can have one or two children. If a node has two children, then the two outgoing edges are labeled by 0 and 1 respectively. Otherwise, the node has one child and the outgoing edge can be labeled either by 0 or by 1. From now on, an FBDD means so defined FBDD.

Definition 7.24 In an FBDD, a *partial-path* starting from v_0 and ended with v_1 is a sequence of vertices $[v_0,u_1,\cdots,u_k,v_1]$ such that (v_0,u_1), (u_i,u_{i+1}), and (u_k,v_1) are edges of the FBDD. When the v_0 is the root, and v_1 is the sink, the partial-path is a *path* of the FBDD. A path is a *complete path* if the set of variables $\{x_1,\cdots,x_n\}$ are all used to label the vertices in the path. A vertex v_1 is the descent node of v_0 if there is a partial path from v_0 to v_1.

The literals associated with the edges in the path form a product term, which is called the *associated product term* of the path. A complete path has a minterm as the associated product term.

Definition 7.25 The *p-indegree* of a node v in an FBDD is defined to be the number of partial-paths from the root of the FBDD to v. The node v is *shared* if its p-indegree is greater than 1. The *p-outdegree* of a node v is the number of partial-paths from v to the sink. The node v is *nontrivially shared* if both its p-indegree and p-outdegree are greater than 1.

Every node in an FBDD has a partial path reaching the sink. Moreover, every node is reachable from the root of the FBDD. Otherwise this node can be deleted without affecting the function represented by the graph. Therefore the p-indegree and p-outdegree of any node in an FBDD has to be greater than 0. If a node v has p-outdegree equals to 1, then all its descent nodes have p-outdegree 1.

Proposition 7.7 For a sparse function f, the paths in its FBDD are all complete paths. Such FBDDs are called sparse FBDDs.

Proposition 7.8 If a sparse FBDD contains a nontrivially shared node v , then the function represented by the graph contains at least two pair of minterms (a_1, a_2) and (b_1, b_2) such that $a_1 - a_2 = b_1 - b_2$.

Proof If the node v is nontrivially shared, then there are at least two partial path U_1 and U_2 from the root v_0 to v, and there are at least two paths U_3 and U_4 from v to the sink. Since the FBDD is sparse, therefore, the connection of U_1 and U_2 with U_3 and U_4 form four complete paths (U_1U_3, U_1U_4, U_2U_3, U_2U_4). These four paths represent four minterms, which are denoted by U_1U_3, U_1U_4, U_2U_3, U_2U_4. We claim that $|U_1U_3 - U_1U_4| = |U_2U_3 - U_2U_4|$.

First we have that the set of variables in U_1 and U_2 must be the same, because they both can be connected to U_3 to form a complete path. The same result holds for the set of variables in U_3 and U_4. Therefore $|U_1U_3 - U_1U_4|$ is equivalent to letting the set of

variables in U_1 be 0. The same result holds for $|U_2U_3 - U_2U_4|$, therefore we have $|U_1U_3 - U_1U_4| = |U_2U_3 - U_2U_4|$.

This proposition would be true for non-sparse FBDDs. However, we do not prove this result here. This proposition shows that BDDs are based on identical difference lists.

Corollary 1 The FBDD of $h(k)$ does not contain any nontrivially shared node.

Proposition 7.9 The FBDD of $h(k)$ has exponential size.

Proof Any FBDD of $h(k)$ contains $2^{\lfloor \frac{k}{2} \rfloor}$ complete paths because each complete path represents a minterm. Let Q denote all the paths of the FBDD, V denote all the nodes in the FBDD, we define a map $M: Q \rightarrow V$ such that for any node $v \in V$, there are at most two paths $Q_1 \in Q$, $Q_2 \in Q$, and $M(Q_1) = M(Q_2) = v$. Therefore $|V| \geq |M(Q)| \geq 2^{\lfloor \frac{k}{2} \rfloor - 1}$, where $|V|$ is the number of elements in the set V.

For a path P, if P does not have any node with p-indegree greater than 1, let the last node before the sink P be v_0, we define $M(P) = v_0$. Since the p-indegree and p-outdegree of v_0 are both 1, P is the only path such that $M(P) = v_0$.

If the node v_1 is the first node in P with p-indegree greater than 1. The parent of v_1 in P is v_0. Then we define $M(P) = v_0$. We claim that at most there is one more path P' such that $M(P') = v_0$.

Suppose $M(P') = v_0$, and v_2 is the first node in P' with p-indegree greater than 1, then v_2 is the child of v_0. Since any FBDD for $h(k)$ does not have any nontrivially shared node, therefore the p-outdegree of v_1 and v_2 is 1, i.e., there is only one partial path from v_1 (v_2) to the sink. Moreover, there is only one partial path from the root to the node v_0, therefore P and P' are the only two paths contain the node v_0.

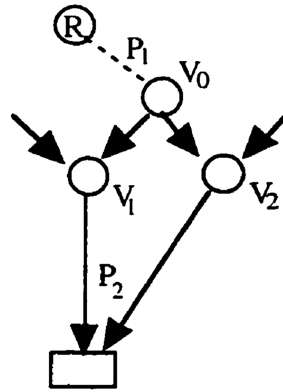


Figure 7.5 Two paths share node v_0

§7.5 Operations on D-Lists

We have defined many integer list data structures such as the I-form and binary graphs, which can represent Boolean functions. In this section, we focus on the default list representation of Boolean functions and derive Boolean operation rules for difference lists. In order to do so, we have to formalize the D-list representation. Then basic operations for D-lists are defined. Operations for OBDDs are special cases of those basic operations.

Definition 7.26 A D-list L is defined recursively by the following equation:

$$L = \varnothing$$
$$L = \{s\}, s \geq 0$$
$$L = L_1 @ L_2$$
$$L = n * L_1$$

For a D-list $L = \{m_1, \cdots, m_n\}$, the *associated list* of L, denoted by $\overline{D}$(L), is the monotonic list with L as the first order difference list, which can be obtained by the operator $\overline{D}$ defined in Table 1, where the operator $\hat{+}$ is defined as $m \hat{+} \{L(1), \cdots, L(n)\} = \{m + L(1), \cdots, m + L(n)\}$. $\overline{D}(L) = \{m_1, m_1 + m_2, \cdots, \sum_{i=1}^{n} m_i\}$. The *range* is the maximum integer of $\overline{D}$(L), i.e., the upper bound, which can be obtained by the operator R also defined in Table 1. Recall the difference operator D maps a monotonic list $L = \{m_1, \cdots, m_n\}$ to the

difference list $D(L) = \{m_1, m_2 - m_1, \cdots, m_n - m_{n-1}\}$. We have $D(\overline{D}(L)) = L$ and $\overline{D}(D(L)) = L$.

Table 7.1

$R: L \rightarrow N$ $R(\varnothing) = 0$ $R(\{s\}) = s$ $R(L_1 @ L_2) = R(L_1) + R(L_2)$ $R(n * L_1) = nR(L_1)$	$\overline{D}: L \rightarrow \Lambda$ $\overline{D}(\varnothing) = \varnothing$ $\overline{D}(\{s\}) = \{s\}$ $\overline{D}(L_1 @ L_2) = \overline{D}(L_1) @ (R(L_1) \hat{+} \overline{D}(L_2))$ $\overline{D}(n * L_1) = \overline{D}((n-1) * L_1) @ ((n-1)R(L_1) \hat{+} \overline{D}(L_1))$ for n>1

An integer n_1 can be used to cut the associated list $\overline{D}(L)$ of a D-list L into two lists: one consisting of integers less than n_1, one consisting of integers greater than n_1. Their corresponding difference lists can be obtained directly from L by the operators $I(n_1, L)$ and $S(n_1, L)$.

Definition 7.27 For a list L and an integer n_1, the list $I(n_1, L)$ and $S(n_1, L)$ are two lists obtained by the following rules.

Table 7.2

$I: N \times L \rightarrow L$ $I(n_1, \varnothing) = \varnothing$, $I(n_1, \{s\}) = \{s\}$, if $s \le n_1$. $= \varnothing$, otherwise. $I(n_1, L_1 @ L_2) = I(n_1, L_1)$, if $n_1 \le R(L_1)$. $= L_1 @ I(n_1 - R(L_1), L_2)$, if $R(L_1) < n_1 \le R(L_1) + R(L_2)$ $= (L_1) @ (L_2)$, otherwise. $I(n_1, n * L) = (k-1) * L @ I(n_1 - (k-1)R(L), L)$, where $(k-1)R(L) \le n_1 < kR(L)$.	$S: N \times L \rightarrow L$ $S(n_1, \varnothing) = \varnothing$ $S(n_1, \{s\}) = \{s\}$, if $s > n_1$ $= \varnothing$, otherwise. $S(n_1, L_1 @ L_2) = \varnothing$, if $n_1 > R(L_1) + R(L_2)$ $= S(n_1, R(L_1) @ L_2)$, if $R(L_1) < n_1 \le R(L_1) + R(L_2)$ $= S(n_1, L_1) @ L_2$, if $n_1 \le R(L_1)$ $S(n_1, n * L) = S(n_1, \{(k-1)R(L)\} @ L) @ ((n-k) * L)$ where $(k-1)R(L) \le n_1 < kR(L)$.

$I(n_1, L)$ and $S(n_1, L)$ can be obtained in linear time in terms of the length of the list L, be it in the I-form or in graphic representation. The essential idea for defining $I(n_1, L)$ is that if the first element *L(1)* of L is less than n_1, then *L(1)* must be in $I(n_1, L)$. At the same time, we need to find the tail list as I({L(2), ..., L(n)}, n_1-L(1)}. The following is an example of finding $I(n_1, L)$. $S(n_1, L)$ is similar.

Example 7.21 If we have the D-list {1, 2, 3, 4}, which represents the monotonic list {1, 3, 6, 10}, then I({1, 2, 3, 4}, 5) = {1}@I({2,3,4},4)={1, 2}@I({3, 4},2)={1, 2}.

Proposition 7.10 Let $[0,n_1]$ be the set of integers less than n_1 and greater than 0, $[n_1+1,¥]$ the set of integers greater than n_1, then we have the following equations.

$$I(n_1,L)@S(n_1,L)=L;\ \overline{D}(I(n_1,L))=\overline{D}(L)\cap[0,n_1];\ \overline{D}(S(n_1,L))=\overline{D}(L)\cap[n_1+1,\infty].$$

The operators $I(n_1,L)$ and $S(n_1,L)$ are the key to defining Boolean operations between lists $L_1 @ L_2$ and $L_3 @ L_4$. Boolean operations (conjunction $\wedge$ and disjunction $\vee$) are interpreted as set operations (union and intersection) between integer lists. For two lists of the form $L_1 @ L_2$ and $L_3 @ L_4$, their associated lists can be similarly represented as $L'_1 @ L'_2$ and $L'_3 @ L'_4$ such that L'_i has L_i as the difference list. Boolean operations between $L_1 @ L_2$ and $L_3 @ L_4$ are actually set operations between $L'_1 @ L'_2$ and $L'_3 @ L'_4$. Moreover, we want to maintain the monotonic property of the result list of set operations between $L'_1 @ L'_2$ and $L'_3 @ L'_4$. Suppose L'_i is bounded by $R(L_i)$. Therefore, all integers in L'_1 are less than $R(L_1)$, all the integers in L'_2 are greater than $R(L_1)$ and less than $R(L_1)+R(L_2)$. In order to perform set union or intersection between $L'_1 @ L'_2$ and $L'_3 @ L'_4$ and maintain the monotonic property at the result list, we need to reorganize the list $L'_1 @ L'_2$ into the form $L''_1 @ L''_2$ such that all the integers in L''_1 are less than $R(L_3)$, all the integers in L''_2 are greater than $R(L_3)$. Therefore we have to use the number $R(L_3)$ to cut the list $L'_1 @ L'_2$. We can do so using the operators $S(n_1,L)$ and $I(n_1,L)$ on the difference list $L_1 @ L_2$ directly. Boolean operations can be performed when we recursively apply the above idea. This procedure is formally described in the following proposition.

Proposition 7.11 For two D-lists of the form $L_1 @ L_2$ and $L_3 @ L_4$ and a Boolean operator OP (conjunction $\wedge$ or disjunction $\vee$), we have the following recursive algorithm.

$$OP(L_1 @ L_2, L_3 @ L_4) = (OP(I(R(L_3), L_1 @ L_2), L_3)) @ (OP(S(R(L_3), L_1 @ L_2), L_4))$$

If the integers in L_1 and L_3 are bounded in the same intervals, i.e., $R(L_3)=R(L_1)$, then we have $I(R(L_3), L_1 @ L_2) = L_1$ and $S(R(L_3), L_1 @ L_2) = L_2$. Therefore the operation rule can be simplified as $OP(L_1 @ L_2, L_3 @ L_4) = (OP(L_1, L_3)) @ (OP(L_2, L_4))$.

A n-variable OBDD graph G_1 represents a function of the form $\bar{x}_1 f_{\bar{x}_1} + x_1 f_{x_1}$.Under the standard integer map, all the minterms in the set $\bar{x}_1 f_{\bar{x}_1}$ are less than 2^{n-1}. Therefore, G_1 can be represented by a D-list of the form $G'_1 = L_1 @ L_2$, where L_1 correspondes to the D-list of $\bar{x}_1 f_{\bar{x}_1}$, and $R(L_1) = 2^{n-1}$. If there is another n-variable OBDD graph G_2 represents a function of the form $\bar{x}_1 g_{\bar{x}_1} + x_1 g_{x_1}$, its corresponding D-list is of the form $G'_2 = L_3 @ L_4$, where $R(L_3) = 2^{n-1}$. Therefore $R(L_3) = R(L_1) = 2^{n-1}$, based on the operation rules for the D-lists, we can derive the OBDD operation rules as special cases.

Similarly, we can define an operation to detect the equality of two D-lists using the I and S operators. When the lists are represented in binary graphs, the complexity of the operations are polynomial in the size of the graphs.

The base case of the recursive algorithm is defined as follows.

Table 7.3

$\wedge : L \times L \to L$	$\vee : L \times L \to L$
$\wedge(\varnothing, L) = \varnothing$	$\vee(\varnothing, L) = L$
$\wedge(s_1, s_2) = \varnothing$, if $s_1 \neq s_2$	$\vee(s_1, s_2) = (s_1) @ (s_2 - s_1)$, if $s_1 < s_2$
$\wedge(s_1, s_2) = s_1$, if $s_1 = s_2$	$\vee(s_1, s_2) = (s_2) @ (s_1 - s_2)$, if $s_1 > s_2$
	$\vee(s_1, s_2) = s_1$, if $s_1 = s_2$

§7.6 Conclusion

In this chapter, we studied integer list representation and Boolean function representation. Many data structures for integer lists are developed. Boolean functions are treated as monotonic lists or piecewise monotonic lists. Integer list representation provides a unified framework for data structures for Boolean functions, which is more flexible and compact as well. If a function has a polynomial size BDD and SOP, then it has a polynomial size difference lists. On the other hand, examples are found with exponential size SOPs and BDDs and with constant size second-order difference lists. Boolean operations are well defined for Boolean functions represented in D-list forms.

The results also explain why BDDs do not work well in general cases. BDDs are basically a method to share identical sublists in the difference lists of Boolean functions. If a difference list is a monotonic list, then it does not contain any identical sublists. In this case, the BDDs tends to be big, and we need new methods such as the second-order difference lists to represent the functions.

Chapter 8 Conclusion

This thesis studies the three problems identified in Chapter 1, i.e., the data structures, minimization, and complexity of Boolean functions.

The Boolean function minimization problem is re-defined. In the minimization problem, Boolean functions have been treated as permutation equivalent classes instead of single functions. The OBDD minimization problem is reformulated. It is proven that functions in a permutation equivalent class have the same minimal OBDD. Therefore for symmetric functions, OBDD minimization is not needed. Moreover, based on a new classification theory, new algorithms for Boolean function minimization are proposed.

A new Boolean function classification theory is proposed. This classification generalizes the traditional supporting variable concept. Supporting variables are distinguished as single-faced variables and double-faced variables. Single-faced variables have similar behavior as redundant variables. Functions are classified as single-faced functions and double-faced functions. Furthermore, it is proven that except for the odd and even parity functions, all other double-faced functions contain some single-faced function restrictions Therefore single-faced functions commonly occur.

New data structures for Boolean functions have been proposed in this thesis. In Chapter 2, the ROSOP form, as a special SOP form, was proposed. In Chapter 7, the difference lists (D-lists), was proposed. Moreover, the newly proposed D-lists can unify many existing data structures for Boolean functions. This approach has many potential applications in discrete domains. Many different representation for integer lists generalized known techniques for data compression.

Many algorithms proposed in this thesis. New minimization algorithms for SOP minimization, factored form minimization, and OBDD minimization, a new algorithm for

detecting single-faced variables, was proposed in Chapter 4. In Chapter 5, a new algorithm for symmetry detection was proposed.

New complexity results are proven. In Chapter 4, it was proven that the optimal OBDDs of complete single-faced functions are of linear size. In Chapter 5, it was proven that the OBDDs of symmetric functions have lower bound $\Omega(n^2)$.

The structure of the OBDD of symmetric functions are shown in Chapter 5, which is totally determined by its left-tail function and its right single-faced function. New implementation and experiment results are also shown in Chapter 5, where the symmetry detection algorithm was implemented and the experiment results were reported.

References

[1] R. Bryant, “Graph-Based Algorithm for Boolean Function Manipulation”, IEEE Trans. on Computers, Vol. C-35, No. 8, August 1986, pp. 677-691.

[2] R. Bryant, “On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication”, IEEE Trans. on Computers, Vol. C-40, No. 2, February 1991, pp. 205-213.

[3] R. Bryant, “ Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams”, ACM Computing Surveys, Vol. 24, No. 3, September 1992.

[4] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, “Logic Minimization Algorithms for VLSI Synthesis", Boston, MA: Kluwer Academic, 1984.

[5] R. K. Brayton, G.D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multi-level Logic Synthesis”, Proceedings of the IEEE, Vol. 78, No. 2, February 1990.

[6] C. Halatsis and N. Gaitanis, "Irredundant Normal Forms and Minimal Dependence Sets of a Boolean Function", IEEE Trans. on Computer., Vol. C-27, No. 11, November 1980, pp. 1064-1068.

[7] A. Mukhopadhyay, "Recent Developments in Switching Theory", Academic Press, 1971.

[8] S. Devadas, “Comparing Two-Level and Ordered Binary Decision Diagram Representations of Logic Functions”, IEEE Trans. Computer-Aided Design, Vol. 12, No. 5, May 1993, pp. 722-723.

[9] S. Chakravarty, "A Characterization of Binary Decision Diagrams", IEEE Trans. on Computers, Vol. 42, No. 2, February 1993.

[10] M. Heap and M. Mercer, "Least Upper Bounds on OBDD sizes", IEEE Trans. on Computers, Vol. 43, No. 6, June 1994

[11] I. Wegener, "The Size of Reduced OBDD's and Optimal Read-once Branching Programs for Almost All Boolean Functions", IEEE Trans. on Computers, Vol. 43, No. 11, November, 1994

[12] J. Gergov and C. Meinel, "Efficient Boolean Manipulation with OBDD's can be Extended to FBDD's", IEEE Trans. on Computers, Vol. 43, No. 10, October 1994

[13] H. Liaw and C. Lin, "On the OBDD-Representation of General Boolean Functions", IEEE Trans. on Computers, Vol. C-41, No. 6, June 1992, pp. 661-664.

[14] I. Wegener, "The Complexity of Boolean Functions", Wiley-teubner series in computer science. New York: Wiley. 1987.

[15] R. Bryant, Y. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams", DAC-95.

[16] D. Sieling and I. Wegner, "Graph driven BDDs - a New Data Structure for Boolean Functions", Theoretical computer science, 1994.

[17] C. L. Berman, "Circuit Width, Register Allocation, and Ordered Binary Decision Diagrams", IEEE Trans. Computer-Aided Design., Vol. 10, No. 8, August 1991, pp. 1059-1066.

[18] K. M. Bulter, D.E. Ross, R. Kapur and M. R. Mercer, "Heuristics to Compute Variable Orderings for Efficient Manipulation of Ordered Binary Decision Diagrams", Proc. of 28th Design Automation Conf. pp.417-420, 1991.

[19] S. J. Friedman and K. J. Supowit, "Finding the Optimal Variable Ordering for Binary Decision Diagrams", IEEE Trans. on Computer., Vol. C-39, No. 5, May 1990, pp. 710-713.

[20] S. Malik, A. Wang, R. Brayton and A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in Logic Synthesis Environment", DAC-88. pp. 624-628.

[21] M. Mercer, R. Kapur and D. Ross, "Functional Approaches to Generating Orderings for Efficient Symbolic Representation", Proc. of 29th Design Automation Conf. pp.40-45, 1992.

[22] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams", Proc. of ICCAD-93, pp. 42-47.

[23] D. I. Cheng and M. Sadowska. "Verifying Equivalence of Functions with Unknown Input Correspondence". Proceeding of EDAC, pp. 81-85, February 1993.

[24] S. L. Hurst. "Detection of Symmetries in Combinatorial Functions by Spectral Means.", Electronic Circuits and Systems, 1(5): 173-180, 1977.

[25] B.G. Kim and D. L. Dietmeyer. "Multilevel Logic Synthesis of Symmetric Switching Functions". IEEE Trans. on CAD. Vol. 10(4):436-446, 1991.

[26] Y.-T. Lai, S. Sastry, and M. Pedram. "Boolean Matching using Binary Decision Diagrams with Applications to Logic Synthesis and Verification". Proceeding of the ICCD92, pp. 452-458, October 1992.

[27] J. Mohnke and S. Malik. "Permutation and Phase Independent Boolean Comparison", Proceedings of EDAC, pp. 86-92, February 1993.

[28] D. Moller, J. Mohnke, and M. Weber. "Detection of Symmetry of Boolean Functions Represented by ROBDDs", ICCAD-93, pp. 680-684.

[29] R. Brayton, " Factoring logic functions", IBM J. Res. Develop. Vol. 31, No. 2, March 1987, pp. 187-198.

[30] M.H. Young and S. Muroga. " Symmetric Minimal Covering Problem and Minimal PLA's with Symmetric Variables", IEEE Trans. on Computers, Vol. C-34, No. 6, June 1985.

[31] E. Sentovich, et al, "SIS, A System for Sequential Circuit Synthesis", Electronics Research Laboratory, Memorandum No. UCB/ERL M92/41. May, 1992.

[32] M. R. Garey and D. S. Johnsom, "Computers and Intractability: a Guide to the Theory of NP-completeness. New York: Freeman, 1979.

[33] Y. Wang, "New Data Structures for Boolean Functions", Technique Report of University of Saskatchewan, May 1994.

[34] Y. Wang, "A New Boolean Function Classification Theory" , Technique Report of University of Saskatchewan, June 1994.

[35] P. Ashar, A. Ghosh and S. Devadas, "Boolean Satisfiability and Equivalence Checking Using General Binary Decision Diagrams", Proceeding of ICCD 1991, pp. 259-264.

[36] J. Burch, "Using BDDs to Verify Multipliers", Proceeding of DAC 1991, pp. 408-412.

[37] S. Jeong, B. Plessier, G. Hachtel and F. Somenzi, "Extended BDD's: Trading off Canonicity for Structure in Verification Algorithms", Proceeding of ICCAD 1991, pp. 408-412.

[38] A. Shen, S. Devadas, and A. Ghosh, "Probabilitstic Construction and Manipulation of Free Boolean Diagrams", ICCAD-93, pp. 544-549.

[39] A. Natapoff, "Irreducible Topological Components of an Arbitrary Boolean Truth Function and Generation of Their Minimal Coverings", J. of ACM, Vol. 14, No. 2, April 1967, pp. 376-381.

[40] D. Goldberg, "Computer Arithmetic", Appendix A in [41], 1990.

[41] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann Publisher, 1990.

[42] S. Panda, F. Somenzi, and B. Plessier, "Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams", ICCAD-94, pp. 628-631.

[43] C. Lin, M. Marek-Sadowska, and D. Gatlin, "Universal Logic Gate for FPGA Design", ICCAD-94, pp. 164-168.

[44] R. I. Bahar, E.A. Frohm, G. M. Gaona, D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic Decision Diagrams and Their Applications", ICCAD-93, pp. 188-191.

[45] A. Srinivasan, T. Kam, S. Malik and R. Brayton, " Algorithms for Discrete Function Manipulation ", Proceeding of ICCAD 1990. 92-95.

[46] A. Friedman and P. Menon, "Theory and Design of Switching Circuits", Computer science press, 1975.

[47] S. Friedman. "Data Structures for Formal Verification of Circuit Design", Ph. D. thesis, Dept. of Computer Science, Princeton Univ. Jan. 1990.

[48] Y. Wang and C. McCrosky, "A New Algorithm for Miniterm-based Logic Minimization", in prepatation.

[49] G. Held, "Data Compression-Techniques and Applications Hardware and Software Considerations", John Wiley & Sons, 1983.